Au cœur du système

- Concepts généraux
- L'API WIN32
- GDI & Programmation Graphique
- Les DLL 32 bits
- Création de Composants
- Communication sous TCP/IP
- Création d'applications INTERNET
- Les ActiveX
- Les communications entre applications
- Utilisation des pilotes ODBC
- Les Threads

Module Programmation Avancée

De plus en plus, DELPHI s'impose comme un outil professionnel de développement dans les entreprises. Ce tome est dédié plus particulièrement aux développeurs désirant concevoir des applications sophistiquées en environnement Windows 32 bits.

La richesse et le succès de DELPHI sont sans nul doute dus au fait que rien n'est impossible à réaliser avec ce produit. C'est l'outil à tout faire, l'outil universel.

L'objectif de ce tome est de donner aux stagiaires des notions de programmation système (accès au système, création de composants, communications)

Sommaire

CONCEPTS GÉNÉRAUX

Delphi pour une programmation RADieuse

*RAD signifie Développement rapide d'application (*Rapid Application Development*). Ce terme décrit la nouvelle génération d'environnements de développement logiciel. Dans un environnement RAD, les programmeurs utilisent des outils plus intuitifs et visuels. Il est difficile de regarder un bout de code qui crée une fenêtre et de la visualiser, mais le RAD permet de créer la fenêtre en quelques clics.*

Avantages de Delphi

Delphi apporte une grande souplesse au développeur. Lorsque Delphi génère un fichier .EXE, il s'agit d'un vrai exécutable. Aucun autre fichier n'est nécessaire pour l'exécution. Vous obtenez donc une application plus propre, et plus facile à distribuer et à maintenir. Vous n'avez à distribuer qu'un seul fichier, sans dépendre de DLL ou d'autres fichiers.

Delphi vous offre donc un compilateur optimisé qui vous donnera une application rapide sans qu'il soit nécessaire de fournir plus d'efforts pour optimiser le programme qu'il n'en avait fallu pour l'écrire.

Les différences entre Delphi 3 et Delphi 2

Bien que l'EDI de Delphi 3 semble inchangé, on compte de nombreuses différences cachées sous le capot. Les principales améliorations sont les suivantes :

- Architecture des bases de données et connectivité. L'architecture des bases de données a été complètement revue afin de suivre une approche multi-liaisons plutôt que la traditionnelle conception Client/Serveur. Vous pouvez ainsi créer des applications client extrêmement légères. Le support natif des bases de données Access a enfin été intégré, afin de permettre une transition aisée des applications VB sous Delphi 3.
- Contrôles ActiveX. Vous pouvez créer vos propres contrôles ActiveX ou utiliser des contrôles déjà écrits dans vos applications Delphi.
- Applications Web. Vous pouvez créer des applications client ou serveur, dans un environnement Web.

Ceci donne à Delphi 3 de nombreux atouts dans la course à Intranet.

RAPPEL SUR LES DIFFÉRENTS TYPES DE BASE

Regardons quels sont les différents "types" de données et l'emploi qu'en fait Delphi 3 :

- Types entiers
- Types réels
- Le type Currency
- Types booléens
- Types caractère
- Types chaîne
- Types variant

TYPES ENTIERS

Le type de données entier est utilisé pour représenter des nombres entiers. Il existe plusieurs types capables de stocker une valeur entière.

Type	Intervalle de valeur	Octets nécessaires	Peut contenir un nombre négatif
Byte	0..255	1	Non
Word	0..65535	2	Non
ShortInt	-128..127	1	Oui
SmallInt	-32768..23767	2	Oui
Integer	-2147483648..2147483647	4	Oui
Cardinal	0..2147483647	4	Non
Longint	-2147483648..2147483647	4	Oui

Vous pouvez remarquer que différents types entiers peuvent avoir des capacités de stockage radicalement différentes. Remarquez aussi que tout a un prix. La quantité de mémoire nécessaire augmente avec la capacité de stockage.

Integer est l'un des deux types génériques de Delphi 3. Les types *génériques* sont ceux qui sont affectés par l'unité centrale ou le système d'exploitation sur lesquels le compilateur est implémenté. Sous un système d'exploitation 32 bits tel que Windows 95 ou Windows NT, les types génériques ont leur capacité de stockage définie par le système d'exploitation.

TYPE RÉELS

Après le type entier, vient logiquement le type de données real (réel). Ces types de données real sont conçus pour contenir un nombre avec une partie fractionnelle.

Type	Intervalle	Octets nécessaires
Real	$\pm$ 2.9 * 10^{-39} à $\pm$ 1.7 * 10^{38}	6
Single	$\pm$ 1.5 * 10^{-45} à 3.4 * 10^{38}	4

Double	$\pm$ 5.0 * 10^{-324} à 1.7 * 10^{328}	8
Extended	$\pm$ 3.4 * 10^{-4932} à 1.1 * 10^{4392}	10
Comp	- 2^{63} à 2^{63} – 1	8

L'intervalle des valeurs que ces types peuvent accepter est impressionnant. Vous avez de quoi faire avant d'avoir besoin de générer un nombre qui dépasse 1.1 * 10^{4392}.

Le type Comp *est en fait un très grand nombre entier, et non un nombre réel. Ce type est inclus dans ce tableau parce qu'il est mis en œuvre de la même façon que les types à virgule flottante. Il s'agit en fait d'un entier à 64 bits.*

Essayez dans la mesure du possible d'utiliser des types de données Single *ou* Double *plutôt que* Real*. Les* Real *sont plus lents à manipuler (il ne s'agit pas d'un type natif pour l'unité à virgule flottante du microprocesseur, chaque opération nécessite donc des conversions) et prennent plus de place ou sont moins précis que* Single *ou* Double*.*

TYPE CURRENCY

Un nouveau type de données qui mérite que l'on s'y intéresse est le type Currency (devise). Jusqu'à présent dans la plupart des langages, le développeur devait utiliser un type Real pour représenter des valeurs monétaires. Delphi 3 propose à cet usage spécifique un type Currency. Ce type est un type à virgule flottante qui est compatible dans ses affectations avec tous les autres types à virgule flottante, type Variant compris (nous reviendrons sur ce type par la suite). Le type Currency a une précision à quatre décimales et on le stocke sous forme d'entier à 64 bits (les quatre chiffres les moins significatifs représentent les quatre chiffres situés après la virgule).

Quel est l'intérêt d'un tel type ? Les principaux avantages sont au nombre de deux :

- Le type Currency est plus précis lorsqu'il s'agit de traiter de grands nombres.
- Le type Currency est utilisé dans CurrencyField et dans d'autres composants. Il est compatible avec les types de base de données représentant des devises.

Le type Currency est un type de données à virgule fixe recommandé pour les calculs monétaires. Il est stocké en tant qu'entier scalaire de 64 bits avec les quatre chiffres les moins significatifs représentant implicitement quatre décimales. L'intervalle de valeurs de Currency est compris entre -922 337 203 685 477,5808 et 922 337 203 685 477,5807. Combinés avec d'autres types réels dans des affectations et des expressions, les valeurs de type Currency sont automatiquement graduées en divisant ou en multipliant par 10 000. Puisque les nombres stockés au format Currency sont des représentations exactes, les opérations sur les valeurs Currency ne sont pas sujettes à des erreurs d'arrondi.

TYPES BOOLÉENS

Les type de données booléen sont les plus simples et les plus utilisés qui soient. Les variables de ce type représentent une quantité logique, True (vrai) et False (faux) par exemple. Vous pouvez alors vous demander pourquoi le Tableau suivant dresse la liste des cinq types booléens différents. Ceci s'explique par des raisons de compatibilité. Dans certains cas, Windows exige une valeur booléenne dont la taille est d'un Word. Dans ces cas-là, d'autres types Booléen peuvent être utiles.

Type	Intervalle	Octets nécessaires
Boolean	Booléen à un octet (le plus répandu)	1
ByteBool	Booléen à un octet	1
Bool	Booléen de taille Word	2
WordBool	Booléen de taille Word	2
LongBool	Booléen de taille deux Words	4

A quoi sert un type Boolean ? Il s'applique à tout ce qui peut se décrire par OUI ou NON, VRAI ou FAUX, ARRET ou MARCHE. Une des choses les plus importantes à remarquer est que les variables de type Boolean peuvent accepter les opérateurs and, or et not. Ceci vous permet de les manipuler plus librement.

TYPES CARACTÈRE

Le type Char est sans doute bien connu de ceux d'entre vous qui avez programmé en C ou C++. Les types caractère ont été conçus pour ne stocker qu'un seul caractère. Un caractère a une taille d'un octet. Un rapide calcul vous montrera que 2^8(un octet) donne la possibilité de choisir parmi 256 caractères différents dans une variable de type Char. Si vous consultez la table ASCII, vous verrez qu'il existe des caractères ASCII allant de 0 à 255 (en informatique, on commence généralement à partir de zéro, et non à partir de un).

Une des nouveautés apportées par Delphi 3 est l'ajout (ou plutôt la redéfinition des types caractère). Le type Char est maintenant l'équivalent du type ANSIChar. ANSIChar est toujours un caractère ANSI 8 bits. Par ailleurs, un troisième type de caractère, WideChar, vous permet d'utiliser un type de caractère 16 bits. A quoi bon trois types différents ? Cette fois encore, il s'agit d'une question de compatibilité. Delphi 3 prend en charge le standard Unicode, comme le montre le Tableau suivant. Le type de données WideChar est le résultat de cette prise en charge. Un caractère Unicode utilise les 16 bits du type WideChar. Si vous placez une valeur normale ANSIChar dans une variable de type WideChar, l'octet de poids fort sera zéro et le caractère ANSI sera stocké dans l'octet de poids faible. Bien que Windows NT respecte parfaitement Unicode, ce n'est pas le cas de Windows 95. Si vous écrivez des applications destinées aux deux plates-formes, pensez à utiliser la fonction SizeOf() et ne partez pas du principe que les caractères ne font qu'un octet de long.

Type de caractère	Taille en octets	Contient
ANSIChar	1	1 caractère ANSI
WideChar	2	1 caractère Unicode
Char	1	Pour l'instant égal à ANSIChar.

Souvenez-vous qu'une variable de type char ne peut contenir qu'un seul caractère. Si vous regardez quel est le caractère ASCII numéro 66, vous verrez qu'il s'agit de la lettre "B". Vous pouvez utiliser le signe # pour indiquer à Delphi 3 que vous souhaitez utiliser la représentation décimale d'un caractère plutôt que le caractère lui-même (ex : choix := #66). Le type de données caractère est très utile et nous amène à passer au type suivant, le type de données String (chaîne de caractères).

TYPES CHAÎNE

Le type de données String est plus souvent utilisé que le type Char. Dans Delphi 1, le type String était une concaténation de 255 caractères au maximum. En d'autres termes, il s'agissait d'un tableau de caractères. Delphi 3 gère les chaînes différemment. Le tableau suivant rappelle les quatre types chaîne disponibles sous Delphi 3.

Type de chaîne	Longueur	Contient	Terminaison nulle
ShortString	255	ANSIChar	Non
AnsiString	Jusqu'à environ 3 Go	ANSIChar	Oui
String	255 ou jusqu'à 3 Go	ANSIChar	Oui ou non
WideString	Jusqu'à environ 1.5 Go	WideChar	Oui

Delphi prend en charge les chaînes longues. Cette prise en charge est activée par la directive de compilation $H+. **Cette directive est présente par défaut**. Lorsque cette directive est utilisée, une variable du type String peut contenir une chaîne de longueur quasi illimitée (environ 3 Go).

Là aussi, Borland a donné au développeur le choix entre rester compatible avec Delphi 1.0 et suivre le progrès. Le type String est par défaut (avec la directive $H+ activée) égale au type AnsiString. Le type AnsiString est une chaîne terminée par un null qui est allouée dynamiquement. Le grand avantage d'une variable de ce type est justement son allocation dynamique. A mesure que vous placez des chaînes plus longues dans cette variable, Delphi 3 réalloue la mémoire nécessaire à votre variable. Un autre avantage du type AnsiString est qu'il est déjà terminé par un null. Vous n'avez donc plus à utiliser les anciennes commandes de type StrPCopy() pour effectuer des conversions entre des chaînes de type Pascal (de longueur fixe) et des chaînes à terminaison nulle.

Quel est l'intérêt de ces terminaisons nulles ? Encore une question de compatibilité. Dans la plupart des appels aux routines systèmes, telles que l'API Win32, il est nécessaire de transmettre aux appels des chaînes à terminaison nulle. L'ancienne chaîne Pascal (désormais appelée ShortString) n'avait pas de terminaison nulle et devait être convertie avant d'être utilisée dans un appel API.

Delphi 3 assure la compatibilité avec Delphi 1.0 en proposant le type ShortString. Ce type est équivalent au type String de Delphi 1.0. Vous pouvez encore définir une chaîne de longueur spécifique, même si la directive $H+ est invoquée.

Voyez l'exemple qui suit :

```
{$H+} {Les chaînes longues sont maintenant activées}
var
    NouvelleString : String; { cette chaîne a une terminaison nulle et elle est allouée dynamiquement }
    AncienneString : String[20]; { En définissant la longueur de cette chaîne, Delphi 3 fait automatiquement de AncienneString un type ShortString, dont la longueur maximale est de 20 caractères }
```

Les composants VCL de Delphi 3 utilisent désormais le type AnsiString pour toutes les propriétés et les paramètres d'événements. Cela simplifie vos interactions avec les VCL et les API, en les uniformisant. Ainsi, vos applications interagissent parfaitement avec d'autres.

TYPE VARIANT

Le type variant est un nouveau type de données dont la caractéristique première est de permettre la manipulation de tout type d'objet et spécialement des tableaux de taille variable et sans type.

Le coté pratique du type variant

Pour présenter l'intérêt de ce nouveau type, voici un exemple simple ou la valeur d'une variable sera considéré soit comme un entier, soit comme une chaîne de caractères, suivant son contexte :

```
procedure TForm1.Button1Click(Sender : TObject);
var
  v : variant;
begin
    v := 34;
    edit1.text :=v;
end;
```

- *Vous serez peut être tenté de n'utiliser que des variables de type variant, mais cette souplesse a un coût. Une variable de type variant consomme beaucoup plus de ressources qu'une variable de type standard.*

Les problèmes liés au type variant

Le type variant procède à un transtypage automatique suivant le contexte dans lequel on se trouve.

Pour mieux comprendre ce processus, codons les appels suivants :

```
function somme(a, b : Variant) : Variant;
begin
    result := a + b ;
end ;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   with listbox1 do
   begin
      items.add(somme('3', '4'));    {34}
      items.add(somme(3, 4));        {7}
      items.add(somme('3', 4));      {7}
      items.add(somme(3, '4'));      {7}
   end;
end;
```

Concrètement, un type variant est une structure mémoire de 16 octets, dont un champ représente le type de la variable référencée. Ce type peut être connu en utilisant la fonction VarType.

```
function VarType(const V : Variant) : Integer;
```

La fonction VarType renvoie le code du type du variant donné. La valeur renvoyée est construite à partir des constantes suivantes déclarées dans l'unité System.

constante	Valeur	Description
varEmpty	$0000;	Le variant est à Unassigned.
varNull	$0001;	Le variant est à Null.
varSmallint	$0002;	Entier signé sur 16 bits (type Smallint).
varInteger	$0003;	Entier signé sur 32 bits (type Integer).
varSingle	$0004;	Valeur à virgule flottante à simple précision (type Single).
varDouble	$0005;	Valeur à virgule flottante à double précision (type Double).
varCurrency	$0006;	Valeur à virgule flottante monétaire (type Currency).
varDate	$0007;	Valeur date et heure (type TdateTime).
varOleStr	$0008;	Référence à une chaîne Unicode allouée dynamiquement.
varDispatch	$0009;	Référence à un objet OLE Automation (pointeur d'interface IDispatch).
varError	$000A;	Code d'erreur du système d'exploitation.
varBoolean	$000B;	Booléen sur 16 bits (type WordBool).
varVariant	$000C;	Variant (utilisé uniquement dans les tableaux de variants).
varUnknown	$000D;	Référence à un objet OLE inconnu (pointeur d'interface IUnknown).
varByte	$0011;	Entier non signé 8 bits (type Byte).
VarString	$0100;	Référence à une chaîne Pascal allouée dynamiquement (type AnsiString).
VarTypeMask	$0FFF;	Masque de bit pour l'extraction du code type.
VarArray	$2000;	Bit indiquant un tableau de variants.

Les bits de poids faible du code du type d'un variant (les bits définis par le masque de bit VarTypeMask) définissent le type du variant. Le bit varArray est initialisé si le variant est un tableau du type donné.

Le type d'un variant peut être modifié en utilisant la fonction standard VarAsType.

Voici notre fonction Somme, réécrite pour contrôler la validité des différentes opérations. On peut par exemple décider que dans le cas de deux paramètres de type différent, l'opération doit être réalisée avec le type du premier paramètre, sauf pour les opérations illégales comme l'ajout d'une date à une chaîne de caractères.

```
function somme(a, b : Variant) : variant;
var
  erreur : Boolean;
begin
    erreur := False;
    if VarType(a) <> VarType(b) then
    begin
        case VarType(a) of
            VarBoolean, VarArray : erreur :=True;
            VarString : erreur := VarType(b) in [VarDate, VarBoolean];
        end;
        if erreur then
          raise Exception.Create ('Addition Impossible');

        case VarType(a) of
            VarString : result := a + String(b);
            else
               result := a + b;
        end;
    end
    else
       result := a + b;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    with listbox1 do
    begin
        items.add(somme('3', '4'));    {34}
        items.add(somme(3, 4));        {7}
        items.add(somme('3', 4));      {34}
        items.add(somme(3, '4'));      {7}
    end;
end;
```

COMPOSITION D'UNE APPLICATION DELPHI

A première vue, un programme simple semble n'être composé que d'un fichier de projet et d'une unité. En réalité, d'autres fichiers sont créées pour vous en coulisse à mesure que vous travaillez sur votre programme. Pourquoi se soucier des fichiers créés par Delphi ? Imaginez un peu votre tête si vous effaciez par mégarde un fichier indispensable à l'application critique sur laquelle vous vous échinez depuis des semaines... Dans tous les cas de figure, il n'est

jamais inutile de bien comprendre ce qui compose un projet Delphi et de savoir d'où viennent tous ces fichiers supplémentaires que vous n'avez pas créés, et ce qu'ils font.

Cette partie aborde plusieurs sujets importants, qui pour certains mériteraient qu'on leur consacre plus que quelques lignes. Mais nous ne ferons que tracer les grandes lignes d'un projet Delphi. Si vous avez besoin de plus de détails concernant un sujet particulier, vous pouvez vous reporter à l'aide en ligne ou aux manuels Delphi. Cette section se propose de vous donner une vue d'ensemble, facilement assimilable.

Une fois que vous saurez ce qui constitue un projet Delphi, vous serez à même de gérer des projets. L'EDI de Delphi comprend un Gestionnaire de projet qui vous assiste dans cette tâche. Cependant, la gestion de projet ne se limite pas à l'utilisation d'un menu. Si vous désirez un projet organisé et bien géré, vous devez penser à définir une structure de répertoires apte à stocker votre code, en utilisant des noms appropriés pour les fichiers, les fiches, les composants et les variables. La meilleure méthode consiste à vous organiser dès le début et à vous tenir à cette organisation jusqu'à l'achèvement du projet.

PROJETS

Un projet Delphi est constitué de fiches, d'unités, de paramètres d'options, de ressources, etc. Toutes ces informations résident dans des fichiers. La plupart de ces fichiers sont créés par Delphi à mesure que vous construisez votre application. Les ressources telles que les bitmaps, les icônes, etc. se trouvent dans des fichiers provenant de sources tierces ou sont créées avec les nombreux outils et éditeurs de ressources dont vous disposez. De plus, des fichiers sont également créés par le compilateur. Jetons un bref coup d'œil sur ces fichiers.

Les fichiers suivants sont créés par Delphi à mesure que vous concevez votre application :

- Le fichier projet (.**dpr**). Stocke des informations concernant les fiches et les unités. Le code d'initialisation se trouve aussi là.

- Le fichier d'unités (.**pas**). Stocke du code. Certaines unités sont associées à des formes, d'autres se contentent de stocker des fonctions et des procédures.

- Fichier de fiches (.**dfm**). Ce fichier binaire est créé par Delphi pour stocker des informations concernant vos fiches. A chaque fiche correspond un fichier Unit (.pas). Ainsi, à mafiche.pas est associé un fichier mafiche.dfm.

- Fichier d'options de projet (.**dfo**). Contient les paramètres d'option du projet.

- Fichiers d'informations de paquet (.**drf**). Ce sont des fichiers binaires utilisés par Delphi pour la gestion des paquets.

- Fichier de ressources (.**res**). Ce fichier binaire contient une icône utilisée par le projet. Ce fichier ne doit pas être créé ou modifié par l'utilisateur. Delphi met à jour ou recrée constamment ce fichier.

- Fichiers de sauvegarde (**.~dp, .~df, .~pa**). Ce sont des fichiers de sauvegarde pour les fichiers de projet, de fiches et d'unités, respectivement.

Les fichiers suivants sont créés par le compilateur.

- Fichier exécutable (.**exe**). C'est le fichier exécutable de votre application. Il s'agit d'un fichier exécutable indépendant qui n'a besoin de rien d'autre que lui-même, sauf si vous utilisez des bibliothèques contenues dans des DLL, VBX ou autres.

- Fichier d'objet unité (.**dcu**). Ce fichier est la version compilée des fichiers d'unités (.pas) et sera lié dans le fichier d'exécutable final.

- Bibliothèque de liaison dynamique (ou DLL) (.**dll**). Ce fichier est créé si vous concevez vos propres DLL.

Enfin, voici d'autres fichiers Windows qui peuvent être utilisés avec Delphi.

- Fichiers d'aide (.**hlp**). Ce sont des fichiers d'aide Windows standard qui peuvent être utilisés avec votre application.

- Fichiers graphiques ou d'images (.**wmf, .bmp, .ico**). Ces fichiers sont fréquemment utilisés dans les applications Windows pour leur donner un aspect agréable et convivial.

Le fichier de projet (.dpr) lui-même contient en fait du code Pascal Objet et constitue la partie
principale de votre application qui lance les choses lorsque vous exécutez votre application. Ce qui est amusant, c'est que vous pouvez construire une application Delphi sans jamais être obligé de voir ce fichier. Il est créé et modifié automatiquement par Delphi à mesure que votre application se construit. Le nom que vous donnez à votre fichier de projet sera aussi celui de votre fichier exécutable. Le code ci-après montre à quoi ressemblerait un fichier de projet si vous commenciez un projet sans changer les noms des fichiers ou des fiches.

```
program Project1
uses
        Forms,

Unit1 in 'UNIT1.PAS' {Form1};

{$R *.RES}

begin
        Application.CreateForm(TForm, Form1);
        Application.Run(Form1);
end.
```

*Le mot **program** à la première ligne indique à Delphi qu'il s'agit du code du programme principal. Program sera remplacé par library (bibliothèque) si vous construisez une DLL.*

Ce fichier est géré automatiquement par Delphi. Il n'est pas recommandé de modifier un fichier de projet, à moins de vouloir écrire une DLL ou de faire de la programmation avancée. Cependant vous pouvez, si vous le désirez, voir le code source du projet en sélectionnant Voir | Source du projet.

FICHES

Le mieux, dans un programme Windows, c'est bien la fiche. Avant de commencer à écrire des programmes Windows, nous programmions surtout des applications DOS en C. C est un très bon et très puissant langage, mais il est loin d'être facile ou souple dès qu'il s'agit de s'atteler à la programmation Windows. Nous avons alors suivi des cours de programmation Windows, et nous avons commencé à programmer en C/C++. Il nous fallait des heures pour créer une simple fiche comportant un bouton, et plus d'heures encore pour comprendre ce que nous venions de faire et comment fonctionnait le code. La quantité de code source nécessaire à la création d'une simple fenêtre en C est gigantesque.

En C++, les choses sont un peu plus faciles si vous utilisez un kit de développement, comme OWL (***Object Windows Library***) de Borland ou MFC (***Microsoft Foundation Class***) de Microsoft, mais vous avez tout de même besoin d'une solide connaissance de la programmation orientée objet et du langage lui-même pour savoir ce que vous faites et ce que vous pouvez faire. Même les compilateurs C++ les plus récents ne proposent pas la programmation visuelle à laquelle excelle Delphi. Delphi donne la possibilité de créer de vrais programmes Windows qui détrônent certaines applications développées dans d'autres environnements visuels. Les programmes Delphi tourneront à des vitesses proches de celles des programmes C++. Delphi se charge de la plus grosse partie du travail et vous laisse vous consacrer à la partie de code la plus spécifique à votre application.

Comme vous le savez, un programme Windows est constitué d'une fenêtre ou d'un ensemble de fenêtres, appelées ***fiches*** *dans Delphi. Lorsque vous démarrez Delphi, il crée automatiquement une fiche à votre usage. Les fiches accueillent vos contrôles, composants, etc.*

L'application Delphi (comme du reste la plupart des applications Windows) est centrée autour de la ***fiche***. Bien que d'autres programmes puissent utiliser le concept de fenêtres ou de fiches, pour qu'une application soit entièrement conforme à l'esprit de Microsoft Windows, elle doit respecter les spécifications de Microsoft dans sa disposition et la structure de son aspect. Les informations concernant les fiches Delphi sont stockées dans deux fichiers : les fichiers ***.dfm*** et ***.pas***. Le fichier .dfm contient en fait des informations sur l'apparence, la taille, l'emplacement de votre fiche. Vous n'avez pas besoin de vous préoccuper de ce fichier, Delphi s'en charge, mais il n'est pas inutile que vous sachiez quelle est sa fonction.

Le code de la fiche et le code des contrôles qu'elle contient sont stockés dans le fichier .pas, aussi appelé ***unité***. C'est le fichier sur lequel vous passez le plus de temps lorsque vous écrivez une application Delphi. Chaque fois que vous ajoutez un gestionnaire d'événements pour une fiche, ou que vous double-cliquez sur un contrôle pour y ajouter du code, le fichier .pas est mis à jour et Delphi place le curseur à l'emplacement adéquat, pour que vous complétiez ou modifiiez votre code. Lorsque vous ajoutez d'autres fiches, elles possèdent elles aussi leurs propres fichiers .dfm et .pas.

Une autre chose à savoir concernant les fiches est qu'elles ont des ***propriétés***. Ces propriétés peuvent être définies pour contrôler l'aspect et le comportement de la fiche. Grâce à ces propriétés, vous pouvez modifier la couleur, la taille, l'emplacement de la fiche, indiquer si elle est centrée, placée à un endroit précis, visible, invisible, etc. Une forme comporte aussi un certain nombre de ***gestionnaires d'événements*** (segments de code s'exécutant lorsque des événements spécifiques liés à la fiche surviennent). Vous pouvez inclure des gestionnaires d'événements pour des événements tels qu'un clic de souris ou un redimensionnement.

Nous avons pratiquement tout dit des fichiers qui composent un projet Delphi. Pour la plupart d'entre eux, ces fichiers seront synchronisés par Delphi lorsque vous procéderez à vos mises à jour. ***Il est recommandé de toujours utiliser Delphi pour changer le nom des fichiers ou pour les mettre à jour, afin d'éviter que les fichiers ne soient désynchronisés****. Si vous passez outre, vous risquez de provoquer des erreurs lors du chargement ou de la compilation de vos programmes. Autrement dit, si vous cliquez sur un composant avant de le supprimer, laissez Delphi supprimer lui-même le code associé. Ne le supprimez pas vous-même dans l'éditeur, Delphi se charge très bien de faire le ménage.*

UNITÉS

On compte trois types d'unités : les unités associées à des fiches (elles sont les plus courantes), les fichiers d'unités qui stockent les fonctions et procédures, et les fichiers d'unités permettant de construire des composants.

Les fichiers d'unités sont des fichiers de code source comportant l'extension .PAS*. En travaillant avec Delphi, vous utiliserez sans cesse des unités.*

Examinons ensemble une unité simple associée à une fiche. Le nom de l'unité figure à la première ligne qui suit le mot unit. Au-dessous de l'en-tête unit ce trouve la partie interface qui contient les clauses uses, type et var. Enfin, la partie implémentation contient les fonctions et procédures de vos contrôles (les gestionnaires d'événements), de même que vos propres fonctions, procédures, et de manière générale tout le code qui sera utilisé dans l'unité.
La partie Implémentation peut également contenir une clause uses.

```
unit Unite1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
    TForm1 = class(TForm)
        procedure FormCreate(Sender: TObject);
    private
        { Déclarations privées }
    public
        { Déclarations publiques }
    end;

var
    Form1: TForm1;

implementation
{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
end.
```

Ce code, accompagné du code du fichier de projet, suffit pour créer un exécutable Delphi qui ouvre une fenêtre. Le programme ne fera pas grand-chose de plus, mais c'est un programme Windows fonctionnel dans sa forme la plus simple. Ce code est le code créé par défaut par Delphi lorsque vous commencez un nouveau projet. Il ne se compilera pas si vous le tapez vous-même dans l'éditeur de code sans créer de nouveau projet.

Regardez les noms qui figurent dans la clause uses. Ce sont les noms d'autres unités. Si vous décidez d'écrire une série de fonctions et de procédures utiles, vous pouvez créer votre propre unité, y placer tous vos utilitaires, et compiler cette unité pour en faire usage ultérieurement. Chaque fois que vous souhaitez utiliser votre unité faite maison, il vous suffit d'ajouter son nom dans la clause uses. Examinons ensemble les différentes parties qui composent l'unité ci-dessus.

- L'en-tête de l'unité (**unit**). Un en-tête d'unité identifie le code comme une unité, et il est suivi du nom, et de celui du fichier de l'unité, dont l'extension sera ***.pas***.
- **Interface**. Cette clause marque le début de la partie interface de l'unité, dans laquelle sont déclarés variables, types, procédures, etc. La partie interface détermine ce qui dans cette unité est disponible pour les autres unités et parties du programme. La partie interface s'achève pour laisser la place à la partie implémentation.
- **Uses**. La clause uses indique au compilateur quelles sont les bibliothèques de fonctions et de procédures qui doivent être compilées dans l'exécutable final. Delphi en inclut certaines automatiquement. Si vous écrivez votre propre unité, vous devez vous souvenir d'en inclure le nom dans la clause uses lorsque vous avez besoin des fonctions qu'elle contient.
- **Type**. La partie déclaration de type permet de créer les types définis par l'utilisateur. Ces types peuvent ensuite être utilisés pour définir des variables. Les spécificateurs de visibilité suivent la clause type dans la partie interface. Les spécificateurs suivants sont utilisés pour contrôler la visibilité d'un objet pour d'autres programmes ou objets.
- **Private**. Les déclarations dans cette section sont traitées comme publiques dans le module, mais resteront inconnues et inaccessibles en dehors de l'unité.
- **Public**. Les déclarations placées dans cette partie sont visibles et accessibles en dehors de l'unité.

Les deux spécificateurs suivants sont utilisés pour la création de composants. Nous les mentionnons ici par souci d'exhaustivité.

- **Published**. Utilisé pour créer des composants. Les propriétés publiées sont affichées dans l'Inspecteur d'objets et peuvent être modifiées au cours de la conception.
- **Protected**. Dans un composant, les champs, méthodes et propriétés déclarés comme protégées sont accessibles aux descendants du type déclaré.

Les quatre spécificateurs (Private, Public, Protected et Published) font partie de la définition de classe.

- **Var**. Utilisé pour déclarer les variables et les variables des objets. Dans une unité de fiche, var est utilisé dans la partie interface (Delphi place cette déclaration pour vous) pour déclarer la fiche comme instance de l'objet TForm. Var est également utilisé pour déclarer les variables dans la partie d'implémentation ainsi que dans les procédures et les fonctions.
- **Implementation**. C'est là que toutes les fonctions et procédures déclarées dans la partie interface seront placées. Toute déclaration faite dans cette partie est privée pour l'unité (elle n'est donc pas disponible pour les autres unités). Vous pouvez cependant ajouter une clause uses dans la section implémentation afin d'avoir accès à d'autres unités.
- **{$R *.DFM}.** Dans une unité de fiche, Delphi insère l'entrée {$R *.DFM} à votre place. Cette entrée est très importante car elle lie la forme à son fichier .dfm dont nous avons parlé tout à l'heure. Pour vous éviter des problèmes, ***ne retirez pas*** cette entrée de votre programme,

Le bloc de code suivant s'exécute lorsque votre fiche est créée. C'est là que vous devez placer le code de démarrage qui doit être exécuté lorsque la fiche commence à se charger. Pour créer cette procédure, utilisez l'Inspecteur d'objets pour visualiser le menu Evénements de la fiche, puis double-cliquez sur l'événement OnCreate.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
end ;
```

N'oubliez bien sûr pas d'ajouter end et remarquez le point qui le suit. Cela signifie qu'il s'agit de la fin de l'unité.

```
end.
```

LES POINTEURS

Ne vous culpabilisez pas si vous n'êtes pas un expert des pointeurs. Il faut de la pratique avant de bien les maîtriser. Lorsque vous créez des structures de données en Delphi 3, de la mémoire leur est allouée. Cette mémoire est nécessaire pour contenir les données de votre structure. La plupart des structures de données (tels les enregistrements et les tableaux) peuvent rapidement devenir très grandes. C'est pour cette raison, entre autres, qu'il convient de n'allouer que la mémoire que vous comptez effectivement utiliser.

Une variable pointeur est capable de stocker l'adresse d'une structure de données (enregistrement ou tableau par exemple). Ceci revient à regarder quelque chose dans l'annuaire téléphonique. Vous pouvez donner ce pointeur vers vos données au lieu des les données elles-mêmes à d'autres fonctions et procédures. Quel en est l'intérêt ? Pensez à une

carte. Vous n'apportez pas tout le contenu du magasin à un voisin pour lui conseiller le magasin, vous vous contentez de lui donner l'adresse du magasin ou son "pointeur".

Les procédures font une grande utilisation des pointeurs. Vous utilisez des pointeurs chaque fois que vous faites un appel de fonction et utilisez le désignateur var dans la déclaration de procédure formelle. Lorsque vous passez des paramètres en utilisant var, ce désignateur passe en fait un pointeur vers les données de votre procédure. C'est la raison pour laquelle lorsque vous modifiez les données qui se trouvent dans une variable paramètre **var** dans votre procédure, les données sont modifiées également en dehors de la procédure. Vous modifiez les données extérieures directement, en utilisant ce "pointeur" vers la donnée.

Cette façon de faire a un grand avantage. Si vous n'utilisez pas le désignateur var dans la liste de paramètres formels de votre appel de procédure, chaque fois que vous passez quelque chose à la procédure, Delphi 3 doit effectuer une copie locale des données pour les manipuler. Si vous n'utilisez pas var, vous déclarez en fait "ne touche pas à mon original". Imaginez un peu la mémoire nécessaire à la copie d'un tableau contenant 10 000 enregistrements. Sans pointeurs, le prix à payer est bien lourd.

UTILISER DES POINTEURS

Pour utiliser un pointeur, vous devez d'abord le définir. Prenons un exemple simple de pointeur vers un nombre réel, comme le montre l'exemple suivant.

```
Program ExemplePointeur;

Uses
        Forms;

type
        PointeurReel = ^Real;

var
        P : PointeurReel;

begin
        New (P);
        P^ := 21.6;
        Dispose (P)
end.
```

Vous voyez ici un exemple simple d'utilisation de pointeur. Vous commencez par créer un type appelé PointeurReel. PointeurReel est un pointeur vers un nombre réel. Le chapeau ^ veut dire "pointant vers". Une fois le pointeur défini, vous devez créer une variable de ce type. Vous créez donc la variable P de type PointeurReel. Vous disposez maintenant d'une variable qui est un pointeur vers un nombre réel.

Vous devez commencer par allouer de la mémoire à P. Pour l'instant, P est capable de pointer vers un nombre réel, mais il ne pointe encore sur rien. En utilisant l'appel de procédure New(), vous demandez à Delphi 3 d'affecter un bloc de mémoire capable de contenir un nombre Real,

et de placer cette adresse mémoire dans P. P pointe maintenant vers un emplacement de mémoire contenant un nombre réel. La ligne P^ := 21.6 doit se lire "Définissez l'emplacement sur lequel pointe P comme étant égal à 21.6". Ceci s'appelle *déréférencer* un pointeur. Autrement dit, vous prenez la valeur 21.6 et la placez dans l'emplacement de mémoire sur lequel pointe P.

Une fois que P ne vous sert plus à rien, vous devez utiliser l'appel de procédure Dispose() pour libérer la mémoire sur laquelle pointe P et pour la rendre au pool de mémoire disponible. Vous terminez alors votre programme.

Vous pouvez également utiliser des pointeurs pour pointer sur des objets plus complexes que de simples nombres réels.

En vérité, Windows est rempli de pointeurs. Le plus souvent, les programmes s'échangent des données au moyen de pointeurs. Dans des langages tels que C++ les pointeurs sont à la base de tout. Nous n'avons fait qu'effleurer le sujet et nous pourrions consacrer un livre tout entier aux pointeurs. Delphi a fait de gros efforts pour cacher les pointeurs, mais il viendra un temps où vous ne pourrez plus y couper, tant le gain de productivité peut devenir important dans certaines situations.

LES UNITÉS DE CODE

Une des raisons pour lesquelles le génie logiciel a progressé si lentement dans les temps anciens de l'informatique (il y a bien 10 ans de cela), était que chacun s'acharnait à réinventer la roue chaque fois qu'il fallait développer une nouvelle application. On ne compte plus le nombre de fois où des programmeurs ont écrit des routines de tri à bulles. Cependant, avec l'arrivée de nouveaux outils de développement, une idée se fit lentement jour.

La création de l'unité permit au programmeur d'écrire et de compiler sa routine de tri à bulles dans une unité de code. Cette unité pouvait ensuite être réutilisée et distribuée à d'autres développeurs. Comme l'unité est compilée, les autres développeurs peuvent voir le code sans voir la source et le secret des algorithmes est précieusement gardé.

FORMAT D'UNE UNITÉ

L'unité est construite comme un programme principal Delphi. La forme générale d'une unité est la suivante :

```
Unit LeNomIci;
interface
    Uses ….
    const …
    type …
    var …
    procedure …
    function …
```

```
implementation
        Uses ….
        const …
        type …
        var …
        procedure …
        function …

initialization {facultatif}

finalization {facultatif}

end. {Fin de l'unité}
```

La section interface de l'unité vient d'abord. C'est là que vous définissez les variables, constantes, types ou autres objets que vous souhaitez rendre disponibles au projet ou aux autres unités qui ont inclus dans leur déclaration le nom de votre unité. Cela vous permet d'inclure des structures prédéfinies qui aident le développeur à utiliser votre unité. On place ensuite dans la section interface les en-têtes de toutes les procédures et fonctions mises en œuvre dans l'unité. C'est de cette manière que Delphi 3 sait ce qui est disponible pour l'application dans votre unité.
Maintenant que vous avez rendu vos intentions publiques (dans la section interface), vous implémentez dans la section adéquate les fonctions et les procédures que vous avez décrites dans la section précédente. Là, vous pouvez tranquillement donner la mesure de votre talent en écrivant votre algorithme de chiffrement ultra-secret qui vous ouvrira les portes de la gloire (et de la prison en France, mais c'est une autre histoire).
Dans la section d'implémentation, vous placez les variables, constantes, etc. que les procédures et fonctions de cette section utiliseront. Vous pouvez également créer des procédures et des fonctions qui seront utilisées localement par les procédures et fonctions spécifiées dans la section interface. Enfin, vous implémentez ces fonctions et procédures décrites dans la section interface. La liste des paramètres doit correspondre parfaitement, ou l'unité ne sera pas compilée.

Deux autres sections de l'unité méritent toute votre attention :

La première est la section **initialization**. Vous pouvez y définir des variables et autres, tout comme dans la section interface. Le problème est que comme la section d'interface ne contient pas de zone d'exécutable, vous ne pouvez pas initialiser ces variables en leur affectant une valeur. La section d'initialisation vous permet de le faire. Là, vous pouvez initialiser vos variables, structures d'enregistrement, variables de fichier et de manière générale tout ce qui peut avoir une valeur initiale. Ceci vous permet de tout mettre en place. Vous pouvez également initialiser les variables dans le bloc begin...end qui se trouve à la fin de l'unité. Vous pouvez y définir des variables et autres, tout comme dans la section interface. Le problème est que comme la section d'interface ne contient pas de zone d'exécutable, vous ne pouvez pas initialiser ces variables en leur affectant une valeur. La section d'initialisation vous permet de le faire.

La section **finalization** est l'opposée de la section précédente. Cette section vous permet de faire un peu le ménage avant de refermer l'application. Vous pouvez ainsi fermer des fichiers,

désallouer de la mémoire et autres activités ménagères. Une fois exécutée la section initialization de votre unité, le code de votre section finalization s'exécutera à coup sûr avant la fermeture de l'application. Delphi 3 exécute les sections de finalisation des unités qui ont été utilisées dans l'ordre inverse de l'exécution des sections d'initialisation. Si vous initialisez les unités X, Y puis Z, elles se refermeront dans cet ordre : Z, Y puis X. Ceci est nécessaire si des unités contiennent d'autres unités dans leur déclaration uses. En effet, dans ce cas les unités ainsi dépendantes devront attendre que leurs sections de finalisation s'exécutent.

Il est important que vous compreniez bien l'ordre dans lequel les différentes sections sont exécutées dans une unité. Lorsque votre application démarre, la section d'initialisation commence son exécution, dans l'ordre des noms d'unités qui figurent dans la déclaration Uses du programme principal. A partir de là, le code des unités est exécuté comme s'il était appelé par votre programme principal. Lorsque l'utilisateur ferme votre application, la section de finalisation de chaque unité est appelée, dans l'ordre inverse de celui des unités qui figuraient dans les sections d'initialisation.

Voici un exemple d'unité permettant d'effectuer deux opérations mathématiques simples. Cette unité n'a pas beaucoup d'applications pratiques, mais elle illustre bien la structure et la fonction d'une unité.

```
Unit Maths;
interface
        function AjouterDeuxNombres (Un, Deux : Integer) : Integer;
        function SoustraireDeuxNombres (Un, Deux : Integer) : Integer;
        function MultiplierDeuxNombres (Un, Deux : Integer) : Integer;
        procedure PositiveKarma;

implementation
        function AjouterDeuxNombres (Un, Deux : Integer) : Integer;
        begin
                AjouterDeuxNombres := Un + Deux
        end;

        function SoustraireDeuxNombres (Un, Deux : Integer) : Integer;
        begin
                SoustraireDeuxNombres := Un - Deux
        end;

        function MultiplierDeuxNombres (Un, Deux : Integer) : Integer;
        begin
                MultiplierDeuxNombres := Un * Deux
        end;

        procedure PositiveKarma;
        begin
                …
        end;
end. {de l'unité Maths}
```

Cette unité simple montre bien la forme que prend une unité. Vous avez défini les fonctions et procédures qui sont disponibles pour l'utilisateur de l'unité de la section d'interface. Dans la section implementation, vous créez les éléments que vous avez annoncés dans la section d'interface. Vous verrez que Delphi 3 fait un usage forcené des unités.

Pour appeler cette unité, il vous suffit de l'inclure dans la section Uses de votre programme principal. L'exemple suivant montre un exemple d'appel à notre unité Maths.

```
Program ExempleMaths;

uses
    Maths;
```

Une fois l'unité ajoutée à votre projet, vous pouvez appeler toutes les fonctions qu'elle contient. Pour ajouter une unité à un projet, sélectionnez Fichier | Utiliser Unité, ou bien passer par le Gestionnaire de projet et cliquer sur le bouton Ajouter.

RÉUTILISATION

Les concepts de réutilisation du logiciel et de bibliothèques de composants ont émergé ces dernières années. L'unité est une extension naturelle de cette théorie de la réutilisation. En effet, une unité permet au développeur de créer un ensemble de routines générales qu'il peut mettre de côté pour l'utiliser à sa guise par la suite.

La déclaration Uses vous permet d'inclure vos propres unités dans votre application. Delphi 3 propose un ensemble d'unités standard qui se chargent de fonctions générales, telles que les E/S de fichiers, les formes, les graphismes, les boutons, et bien d'autres encore (une liste complète des unités proposées par Delphi figure dans l'aide en ligne). L'usage d'unités procure plusieurs avantages. Comme en grande partie les fonctionnalités d'une application peuvent être divisées en plusieurs groupes ou zones, il semble logique d'adopter un modèle de programmation qui suive ce concept.

Les unités rendent également plus facile la phase de débogage. Si vous rencontrez une difficulté avec votre formule de maths, il vous suffit de consulter votre unité mathématique pour déboguer la fonction, au lieu de devoir fouiller dans la totalité de votre application pour dénicher l'erreur. La capacité à fragmenter votre programme vous permet de regrouper fonctions et procédures dans des unités et ainsi de mieux organiser votre projet. Dans un projet de taille importante et où de nombreuses personnes sont appliquées, vous pouvez même désigner un bibliothécaire de code chargé de conserver les dernières versions de vos unités et de les distribuer.

DISTRIBUTION ET SÉCURITÉ

Comme l'écriture d'un livre, l'écriture d'un logiciel consiste à créer quelque chose en ne partant de rien (ou presque). Cet effort de création doit être protégé. Si vous découvrez un algorithme de chiffrage révolutionnaire, vous ressentirez vite le besoin de protéger votre code, tout en ayant la possibilité de le vendre. Vous voilà face à un dilemme : comment vendre

votre code à d'autres développeurs sans pour autant leur fournir le code source et risquer ainsi de révéler vos algorithmes ?

Les unités sont un moyen parfait pour ceux qui désirent distribuer leur code sans pour autant l'exposer au piratage. Les unités Delphi peuvent être compilées en fichiers binaires et distribuées sous cette forme. Lorsqu'une unité est compilée, Delphi lui donne un suffixe .DCU. Cela indique qu'il s'agit d'une Unité Compilée Delphi (Delphi Compiled Unit en anglais). Vous pouvez distribuer votre unité sous cette forme, et d'autres personnes pourront utiliser votre unité pour leurs applications (en l'incluant dans la déclaration Uses), sans pour autant voir le code source. Ceci vous permet de développer votre code et de le commercialiser en toute sécurité.

Pour que des développeurs puissent utiliser votre unité, ils doivent connaître les fonctionnalités qu'elle propose. Il est donc nécessaire que vous décriviez en détail ces fonctionnalités dans un document accompagnant l'unité. De nombreux développeurs se contentent de copier la section interface de leur unité pour la distribuer comme documentation (en effet, puisque l'unité est compilée, la section interface n'est plus lisible). Il existe un véritable marché pour les unités, les DLL et les VCL et vous pourriez très bien y prendre pied un jour ou l'autre.
Il convient de préciser toutefois que jusqu'à présent, ce concept de distribution des unités n'a pas toujours bien fonctionné lorsque les versions des produits changeaient. Il est généralement nécessaire de recompiler des unités pour chaque version de Pascal/Delphi. C'est pour cette raison que de nombreux programmeurs mettent à la disposition des acheteurs leur code source (moyennant finances bien sûr).

LA GESTION DES EXCEPTIONS

La complexité d'une application fonctionnant en mode événementiel dans un environnement graphique ne permet pas de pouvoir prévoir, de manière algorithmique, toutes les causes d'erreurs.

Delphi, comme la plupart des environnements de développement modernes, signale les conditions d'erreur par des exceptions. Une exception est un objet contenant des informations indiquant quelle erreur (matérielle ou logicielle) s'est produite et où elle s'est produite.

Pour que les applications soient robustes leur code doit reconnaître les exceptions lorsque ces dernières se produisent et y répondre (sous peine de voir s'afficher un message d'erreur généralement peu explicite). On gère les exceptions, c'est à dire qu'on est capable de les détecter afin d'intervenir sur le déroulement du programme, lorsque l'on souhaite :

- Protéger certains blocs d'instructions ;
- Protéger les allocations de ressources systèmes ;

Il est possible de traiter les exceptions courantes ou de définir ses propres exceptions.

- Cette gestion était déjà réalisable en Pascal, mais cela impliquait l'usage de directive telle que {$I+}/{$I-} (pour capter les problèmes potentiels d'entrée/sortie).

➢ La gestion des exceptions simplifie énormément les problèmes pouvant provenir d'une division par zéro, d'une tentative d'ouverture de fichier non-existant, de lecture sur une disquette non insérée, etc.

Delphi propose deux manières de gérer les exceptions. Il permet aussi de lever une exception de manière dynamique.

Les lignes de code susceptibles de pouvoir créer une exceptions doivent être placées dans un bloc d'instruction (dit ' bloc protégé ') matérialisé par le mot réservé ' Try' et se terminant par un 'end ;'.

➢ Contrairement à l'habitude il n'y a donc pas couplage incontournable de 'begin ' / 'end '. Bien en prendre conscience car cela peut être cause d'erreurs.

Lorsqu'une exception se produit dans le bloc ainsi matérialisé, l'exécution du programme est directement déroutée vers le bloc de 'traitement de l'exception' , matérialisé par les mots 'Except' ou 'Finally '.

- Si le bloc d'instruction contenu dans la partie 'Try' contient plusieurs instructions, celles qui sont comprises entre l'instruction qui a généré l'exception et le bloc de traitement sont ignorées. Attention.....

- Si l'on souhaite voir le résultat du traitement de l'interruption, il faut enlever l'option "Stopper si exception" du menu 'Options | Environnement' onglet 'Préférences'. En effet si cette option est sélectionnée, une exécution du programme dans l'environnement Delphi se traduira par un arrêt du programme lorsque l'exception se déclenchera, malgré la mise en place du traitement.

Try... Except... End

Ce premier type de gestion d'exception permet de travailler avec des commandes susceptibles de générer des exceptions, telles qu'une division par zéro, un débordement de pile, etc.. La syntaxe en est la suivante :

```
Try
        { Code susceptible de générer une exception }
Except
        { Code à exécuter dès qu'une exception s'est produite }
End ;
```

Supposons, que nous voulons créer une fonction permettant de calculer la valeur d'une fonction de type F (X). Le prototype d'une telle fonction peut être :

```
Function F ( X : Real ; Var Y : Real) : Boolean ;
```

Avec Y = F (X), le booléen renvoyé permet de savoir si la fonction F est bien définie pour la valeur donnée de X.
Le problème dans l'écriture de cette fonction est de savoir quel est l'intervalle de définition de la fonction F. Mais à travers la gestion des exceptions, le problème peut être contourné:

```
Function F (X : Real ; Var Y : Real) : Boolean ;
begin
        Result := True ;
        { Par défaut on suppose que F ( X ) est défini }
        Try
                Y := 1 / (X * X - 4 * X + 3) ;
                { Génère une exception pour X = 1 et X = 3 }
        Except
                { Cette ligne n'est exécutée que si F ( X ) n'est pas définie }
                Result := False ; { Result = valeur renvoyée }
        end ;
end ;
```

{ Un ' end ' pour le bloc de gestion des exceptions et un pour la fin de la fonction }

Try... Finally... End

Cette syntaxe est plus particulièrement utilisée lorsque l'on doit travailler sur des ressources dynamiques. L'usage en est le suivant :

```
{ création de la ressource dynamique }
Try
        { Code susceptible de générer une exception }
Finally
        { Libération de la ressource dynamique }
End ;
```

Dans ce cas, la clause Finally est toujours exécutée (qu'il y ait eu exception ou non).

Autrement dit :

- Si dans la clause Try, une ligne de code génère une exception, alors on passe directement à la clause Finally.
- Si aucune exception n'est générée dans la clause Try, alors, une fois la dernière ligne de code du bloc protégé exécutée, on exécute les instructions contenues dans la clause Finally.

Exemple :
L'on souhaite modifier le texte d'un bouton en réaction à un événement " click" dessus.

Le nouveau texte correspond au résultat d'une fonction F(X) définie dans l'exemple précédent. Ce qui peut générer une exception si le calcul de F(X) n'est pas réalisé dans son domaine de définition.

On peut résoudre le problème de la manière suivante :

```
Procedure TForm1 . Button1Click (Sender : TObject) ;
Var
        Resultat : MaClasse ;
Begin
        Resultat := MaClasse . Create ; { Création d'une ressource }
        Try
                Resultat . VarPublique := F ( 3 ) ; { Génère une exception }
                Button1 . Caption := FloatToStr ( Resultat . VarPublique) ;
        Finally
                Resultat . Free ;        { Libère la ressource }
        End ;
End ;
```

Si une exception a lieu, alors le message suivant apparaîtra :

Ce qui est déjà plus sympathique qu'un blocage complet du système après affichage d'une fenêtre système sur fond blanc de mauvais aloi.

Raise

Le mot clé 'Raise' permet de générer une exception , volontairement, à partir du programme. Cela se fait grâce à la syntaxe :

```
Raise <CLASSE D'EXCEPTION> . Create (<MESSAGE>) ;
```

Dans l'exemple précédent, on suppose que l'on connaît le domaine de définition de la fonction. Par conséquent nous pouvons avoir :

```
Function F ( X : Real ; Var Y : Real ) : Boolean ;
Begin
        Result := True ;
        If ((X = 1) Or (X = 3)) Then
        Begin
                Raise EDivByZero .Create (' X doit être différent de ' + '1 et de 3...' ) ;
                Result := False ;
        End
        Else
                Y := 1 / (X * X - 4 * X + 3) ;
End ;
```

Si X vaut 1 ou 3, Raise va provoquer l'exception EDivByZero. Cela se caractérise par l'apparition d'un message Windows :

Les différentes exceptions

Chaque exception est gérée par un objet particulier dérivée de la classe Exception. Delphi propose un nombre relativement important d'objets exceptions chargés de traiter différents types d'exceptions.

Chaque exception pouvant arriver est définie par un identificateur unique qu'il est possible d'utiliser dans la syntaxe : ***On < Exception_Id > do .....***

Il est possible que plusieurs exceptions soient susceptibles de se produire pendant l'exécution du bloc d'instruction "protégé" par Try . On peut alors tester la valeur de l'exception qui s'est produite pour adapter précisément les instructions à exécuter dans le bloc dépendant de Except.

Liste des principales exceptions :

EConvertError	Exception lancée lorsque les fonctions StrToInt ou StrToFloat ne sont pas en mesure de convertir la chaîne spécifiée vers une valeur entière ou flottante valide.
EDataBaseError	Exception déclenchée lorsqu'une erreur de base de données se produit (Par exemple, si l'application tente d'accéder aux données d'une table qui n'est pas encore ouverte)
EDBEditError	Exception déclenchée lorsque les données ne sont pas compatibles avec le masque défini pour le champ.
EDBEngineError	Exception déclenchée quand une erreur BDE se produit.
EDivByZero	Exception liée aux calculs sur les entiers. L'exception se produit lorsque votre application tente une division par 0 sur un type entier.
EFCreateError	Exception déclenchée lorsqu'une erreur se produit à la création d'un fichier (par exemple, le nom du fichier spécifié peut être incorrect ou le fichier ne peut être recréé car il n'est accessible qu'en lecture).

EFOpen Exception déclenchée lors d'une tentative de création 'un objet flux de fichier et le fichier indiqué ne peut être ouvert.

EGPFault Exception liée au matériel déclenchée lorsque l'application tente d'accéder à une partie de la mémoire qui lui est interdite.

EInOutError Exception déclenchée à chaque fois qu'une erreur d'entrée/sortie MS-DOS se produit. Le code d'erreur résultant est renvoyé dans le champ ErrorCode. La directive $I+ doit être activée pour qu'une erreur d'entrée/sortie déclenche une exception. Si une erreur d'E/S se produit alors que l'application se trouve dans l'état $I-, celle-ci doit appeler la fonction IOResult pour résorber l'erreur.

EIntOverFlow Exception liée aux calculs arithmétiques sur des entiers. Elle se produit lorsque le résultat d'un calcul est trop grand pour le registre qui lui est alloué, entraînant la perte de la donnée.

EInvalidGraphic Exception déclenchée lorsque l'application tente d'accéder à un fichier qui n'est pas un bitmap, une icône, un métafichier, ou un graphique défini par l'utilisateur.

EInvalidGridOperation Exception déclenchée lorsqu'une opération non permise est tentée sur une grille (par exemple, lorsque l'application essaie d'accéder à une cellule inexistante).

EInvalidPointer Exception déclenchée lorsque l'application tente une opération non permise sur des pointeurs.

EListError Exception déclenchée lorsqu'une erreur se produit dans un objet liste, chaîne, ou liste de chaînes. Les exceptions liées aux erreurs se produisent si votre application fait référence à un élément se situant en-dehors de la portée de la liste.

EOutOfMemory Exception signalant les erreurs du tas. Elle se produit lorsque l'application tente une allocation dynamique de mémoire et que l'espace mémoire disponible dans le système est insuffisant pour accomplir l'opération demandée.

EOutOfResources Exception se produisant lorsque votre application tente de créer un descripteur Windows et que Windows n'est pas en mesure de fournir un descripteur pour l'allocation

EPrinter Exception déclenchée lorsqu'une erreur se produit à l'impression.

ERangeError Exception liée aux calculs sur les entiers. Elle se produit lorsque l'évaluation d'une expression entière dépasse les bornes admises pour le type entier de l'affectation.

EZeroDivide Exception liée aux calculs sur des flottants. Elle se produit lorsque votre application tente de diviser une valeur flottante par zéro.

Exemple :

```
Try
        { Code susceptible de générer une exception }
Except
        On EDivByZero Do
        { Code à exécuter lorsqu'il se produit une division par zéro }
        On EDDEError Do
        { Code à exécuter s'il y a un problème de communication DDE}
        ...
End ;
```

Dans cette version, on ne prend en compte que certaines exceptions : la gestion est donc plus précise.

L'événement OnException du composant TApplication

Le composant TApplication dispose de l'événement OnException qui permet de gérer toutes les exceptions qui ne sont pas prises en compte par ailleurs et qui, par défaut seraient traitées par la méthode HandleException (une boîte de message est affichée pour indiquer qu'une erreur a eu lieu).

Comme TApplication n'est pas accessible via l'inspecteur d'objet il faut écrire le code suivant:

```
procedure TForm1.GereException ( Sender: TObject);
begin
        MessageDlg ( 'Une erreur s''est produite !!!! ', mtWarning,[mbOk], 0);
end;
```

{ Ne pas oublier de déclarer l'en tête de la procédure dans la section déclaration de l'unité }

L'initialisation du gestionnaire d'événement se fera alors sous la forme :

```
procedure TForm1.FormCreate ( Sender : TObject ) ;
begin
        .......
        Application.OnException := GereException ;
        .....
end ;
```

En conclusion du chapitre 1 : Exemple d'utilisation du type variant

```
unit PileVariant_f;
interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs,StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Déclarations privées }
    public
        { Déclarations publiques }
 end;

var
    Form1: TForm1;

implementation
{$R *.DFM}

const
    MAXPILE = 20;   // nombre maximum d'éléments dans la pile

type
    TPile = class
     private
        Elements : array[0..MAXPILE] of variant;
        Sommet : integer;
    public
        constructor Create;
        function EstVide : boolean;
        function EstSaturee : boolean;
        procedure Empile(v : variant);
        function  Depile : variant;
 end;

constructor TPile.Create;
begin
    Sommet:=-1;        // Une pile est dite vide si le sommet est négatif
end;

function TPile.EstVide : boolean;
begin
    result:=Sommet<0;   // renvoie vrai si le sommet est négatif
```

```
end;
function TPile.EstSaturee : boolean;
begin
	result:=Sommet=MAXPILE; // vrai si toutes les cellules sont remplies
end;

procedure TPile.Empile(v : variant);
begin
	if EstSaturee then
		raise Exception.Create('La pile est saturée');
	inc(Sommet);
	Elements[Sommet]:=v;
end;

function TPile.Depile : variant;
begin
	if EstVide then
		raise Exception.Create('La pile est vide');
	result:=Elements[Sommet];
	dec(Sommet);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
	Pile : TPile;
begin
	Pile:=TPile.Create;
	Pile.Empile('degré');
	Pile.Empile(20);
	Pile.Empile('Il fait');
	Pile.Empile('heure.');
	Pile.Empile(time);
	Pile.Empile('Il est');
	Pile.Empile('Bonjour');
	while not Pile.EstVide do
		ShowMessage(Pile.Depile);
end;

end.
```

Mais que fait donc cette application ?

L'API WIN32

L'interface de programmation API Win32 (Win32 Application Programming Interface) est commune aux systèmes d'exploitation Windows 95 et Windows NT. L'utilisation d'une API commune permet d'écrire des applications pouvant être déployées sur différents systèmes d'exploitation sans aucune modification.

Pour toute information concernant les variables ou les types, se référer au fichier Windows.pas fourni avec DELPHI 3.

FONCTIONS D'ENTRÉE UTILISATEUR

Sous Windows, une application reçoit les entrées de l'utilisateur par l'intermédiaire de la souris ou du clavier.

Fonction	**ClipCursor**
Syntaxe	function ClipCursor(lpRect: PRect): BOOL;
Paramètres	lpRect: PRect
Description	Positionne le curseur sur une surface rectangulaire de l'écran

Fonction	**GetCapture**
Syntaxe	function GetCapture: HWND;
Paramètres	
Description	Récupère le handle de la fenêtre ayant capturée la souris (celle qui reçoit les entrées souris)

Fonction	**GetCursorPos**
Syntaxe	function GetCursorPos(var lpPoint: TPoint): BOOL;
Paramètres	var lpPoint: Tpoint
Description	Récupère la position du curseur en coordonnées écran

Fonction	**SwapMouseButton**
Syntaxe	function SwapMouseButton(fSwap: BOOL): BOOL;
Paramètres	fSwap: BOOL
Description	Inverse la signification des boutons gauche et droit

INTERFACE DE PROGRAMMATION GRAPHIQUE

Fonction	**GetWindowDC**
Syntaxe	function GetWindowDC(hWnd: HWND): HDC;
Paramètres	hWnd: HWND
Description	Extrait le contexte périphérique de toute la fenêtre

Fonction	PtVisible
Syntaxe	function PtVisible(DC: HDC; p2, p3: Integer): BOOL;
Paramètres	DC: HDC; p2, p3: Integer ;
Description	Détermine si un point donné se trouve dans la zone de clipping d'un contexte périphérique

Fonction	RectVisible
Syntaxe	function RectVisible(DC: HDC; const Rect: TRect): BOOL;
Paramètres	DC: HDC; const Rect: Trect ;
Description	Détermine si des parties du rectangle spécifié se trouvent dans la zone de clipping d'un contexte périphérique

Fonction	ReleaseDC
Syntaxe	Function ReleaseDC(hWnd: HWND; hDC: HDC): Integer;
Paramètres	hWnd: HWND; hDC: HDC ;
Description	Libère un contexte périphérique

Fonction	GetBValue, GetGValue, GetRValue

Syntaxe	function GetRValue(rgb: DWORD): Byte;
	function GetGValue(rgb: DWORD): Byte;
	function GetBValue(rgb: DWORD): Byte;

Paramètres	rgb: DWORD
Description	Récupère la composante bleu, vert, rouge d'une valeur RVB

E/S DANS LES FICHIERS

Fonction	CompareFileTime
Syntaxe	function CompareFileTime(const lpFileTime1, lpFileTime2: TFileTime): Longint;
Paramètres	const lpFileTime1, lpFileTime2: TFileTime
Description	Compare deux dates de création de fichiers

Fonction	CopyFile
Syntaxe	function CopyFile(lpExistingFileName, lpNewFileName: PChar; bFailIfExists: BOOL): BOOL;
Paramètres	LpExistingFileName, lpNewFileName: PChar; bFailIfExists: BOOL ;
Description	Copie un fichier existant dans un nouveau fichier

Fonction	**CreateDirectory**
Syntaxe	function CreateDirectory(lpPathName: PChar; lpSecurityAttributes: PSecurityAttributes): BOOL;
Paramètres	lpPathName: PChar; lpSecurityAttributes: PsecurityAttributes ;
Description	Crée un nouveau répertoire

Fonction	**CreateFile**
Syntaxe	function CreateFile(lpFileName: PChar; dwDesiredAccess, dwShareMode: Integer; lpSecurityAttributes: PsecurityAttributes; dwCreationDisposition, dwFlagsAndAttributes: DWORD; hTemplateFile: Thandle): THandle; stdcall;
Paramètres	lpFileName: PChar; dwDesiredAccess, dwShareMode: Integer; lpSecurityAttributes: PsecurityAttributes; dwCreationDisposition, dwFlagsAndAttributes: DWORD; hTemplateFile: Thandle ;
Description	Crée, ouvre, tronque un fichier

Fonction	**DeleteFile**
Syntaxe	function DeleteFile(lpFileName: PChar): BOOL;
Paramètres	lpFileName: Pchar ;
Description	Supprime un fichier

Fonction	**FindFirstFile**
Syntaxe	function FindFirstFile(lpFileName: PChar; var lpFindFileData: TWIN32FindData): THandle;
Paramètres	lpFileName: PChar; var lpFindFileData: TWIN32FindData ;
Description	Parcours un répertoire à la recherche du premier fichier répondant à un critère de recherche

Fonction	**FindNextFile**
Syntaxe	function FindNextFile(hFindFile: THandle; var lpFindFileData: TWIN32FindData): BOOL;
Paramètres	hFindFile: THandle; var lpFindFileData: TWIN32FindData ;
Description	Recherche le fichier suivant répondant à un critère de recherche

Fonction	**GetFileAttributes**
Syntaxe	function GetFileAttributes(lpFileName: PChar): DWORD;
Paramètres	lpFileName: Pchar
Description	Récupère les attributs d'un fichier

Fonction	**GetFileInformationByHandle**
Syntaxe	function GetFileInformationByHandle(hFile: THandle; var lpFileInformation: TByHandleFileInformation): BOOL; stdcall;
Paramètres	hFile: Thandle; var lpFileInformation: TbyHandleFileInformation ;
Description	Récupère les informations concernant un fichier

Fonction	**GetFileSize**
Syntaxe	function GetFileSize(hFile: THandle; lpFileSizeHigh: Pointer): DWORD;
Paramètres	hFile: Thandle; lpFileSizeHigh: Pointer ;
Description	Récupère la taille d'un fichier en octets

Fonction	**GetFileTime**
Syntaxe	function GetFileTime(hFile: THandle; lpCreationTime, lpLastAccessTime, lpLastWriteTime: PFileTime): BOOL;
Paramètres	hFile: Thandle; lpCreationTime, lpLastAccessTime, lpLastWriteTime: PfileTime ;
Description	Récupère la date et l'heure à laquelle un fichier a été créé, consulté et modifié pour la dernière fois

Fonction	**GetFileType**
Syntaxe	function GetFileType(hFile: THandle): DWORD;
Paramètres	hFile: Thandle ;
Description	Récupère le type d'un fichier

Fonction	**GetFullPathName**
Syntaxe	function GetFullPathName(lpFileName: PChar; nBufferLength: DWORD; lpBuffer: PChar; var lpFilePart: PChar): DWORD;
Paramètres	lpFileName: PChar; nBufferLength: DWORD; lpBuffer: PChar; var lpFilePart: Pchar ;
Description	Récupère le chemin ainsi que le nom complet d'un fichier

Fonction	**GetTempPath**
Syntaxe	function GetTempPath(nBufferLength: DWORD; lpBuffer: PChar): DWORD;
Paramètres	NBufferLength: DWORD; lpBuffer: Pchar ;
Description	Récupère le chemin du répertoire chargé de contenir les fichiers temporaires

Fonction	**MoveFile**
Syntaxe	function MoveFile(lpExistingFileName, lpNewFileName: PChar): BOOL;
Paramètres	LpExistingFileName, lpNewFileName: Pchar ;
Description	Déplace un fichier ou un répertoire et son arborescence

Fonction	**ReadFile**
Syntaxe	Function ReadFile(hFile: THandle; var Buffer; nNumberOfBytesToRead: DWORD; var lpNumberOfBytesRead: DWORD; lpOverlapped: POverlapped): BOOL;
Paramètres	hFile: Thandle; var Buffer; nNumberOfBytesToRead: DWORD; var lpNumberOfBytesRead: DWORD; lpOverlapped: Poverlapped ;
Description	Lit les données d'un fichier

Fonction	**RemoveDirectory**
Syntaxe	function RemoveDirectoryA(lpPathName: PAnsiChar): BOOL;
Paramètres	lpPathName: PansiChar ;
Description	Supprime un répertoire vide

Fonction	**SearchPath**
Syntaxe	function SearchPath(lpPath, lpFileName, lpExtension: PChar; nBufferLength: DWORD; lpBuffer: PChar; var lpFilePart: PChar): DWORD;
Paramètres	lpPath, lpFileName, lpExtension: PChar; nBufferLength: DWORD; lpBuffer: Pchar; var lpFilePart: Pchar ;
Description	Recherche un fichier

Fonction	**SetEndOfFile**
Syntaxe	function SetEndOfFile(hFile: THandle): BOOL;
Paramètres	hFile: Thandle ;
Description	Place le caractère de fin de fichier à la position actuelle du pointeur de fichier

Fonction	**SetFileAttributes**
Syntaxe	function SetFileAttributes(lpFileName: PChar; dwFileAttributes: DWORD): BOOL;
Paramètres	lpFileName: PChar; dwFileAttributes: DWORD ;
Description	Spécifie les attributs d'un fichier

Fonction	**SetFilePointer**
Syntaxe	function SetFilePointer(hFile: THandle; lDistanceToMove: Longint; lpDistanceToMoveHigh: Pointer; dwMoveMethod: DWORD): DWORD;
Paramètres	hFile: Thandle; lDistanceToMove: Longint; lpDistanceToMoveHigh: Pointer; dwMoveMethod: DWORD ;
Description	Déplace le pointeur d'un fichier ouvert

Fonction	**SetFileTime**
Syntaxe	Function SetFileTime(hFile: THandle; LpCreationTime, lpLastAccessTime, lpLastWriteTime: PFileTime): BOOL;
Paramètres	HFile: THandle; LpCreationTime, lpLastAccessTime, lpLastWriteTime: PfileTime ;
Description	Spécifie la date et l'heure à laquelle un fichier a été créé, consulté ou modifié pour la dernière fois

Fonction	**SystemTimeToFileTime**
Syntaxe	function SystemTimeToFileTime(const lpSystemTime: TSystemTime; var lpFileTime: TFileTime): BOOL;
Paramètres	const lpSystemTime: TSystemTime; var lpFileTime: TfileTime ;
Description	Convertit la date et l'heure du système en date et heure d'un fichier

Fonction	**WriteFile**
Syntaxe	function WriteFile(hFile: THandle; const Buffer; nNumberOfBytesToWrite: DWORD; var lpNumberOfBytesWritten: DWORD; lpOverlapped: POverlapped): BOOL;
Paramètres	hFile: THandle; const Buffer; nNumberOfBytesToWrite: DWORD; var lpNumberOfBytesWritten: DWORD; lpOverlapped: Poverlapped ;
Description	Ecrit des données dans un fichier

LA BASE DE REGISTRE

Fonction	**RegCreateKeyEx**
Syntaxe	function RegCreateKeyEx(hKey: HKEY; lpSubKey: PChar; Reserved: DWORD; lpClass: PChar; dwOptions: DWORD; samDesired: REGSAM; lpSecurityAttributes: PsecurityAttributes; var phkResult: HKEY; lpdwDisposition: PDWORD): Longint;
Paramètres	hKey: HKEY; lpSubKey: PChar; Reserved: DWORD; lpClass: PChar; dwOptions: DWORD; samDesired: REGSAM; lpSecurityAttributes: PsecurityAttributes; var phkResult: HKEY; lpdwDisposition: PDWORD ;
Description	Crée une nouvelle sous-clé

Fonction	**RegDeleteKey**
Syntaxe	function RegDeleteKey(hKey: HKEY; lpSubKey: PChar): Longint;
Paramètres	hKey: HKEY; lpSubKey: Pchar ;
Description	Supprime une clé d'un registre

Fonction	**RegDeleteValue**
Syntaxe	function RegDeleteValue(hKey: HKEY; lpValueName: PChar): Longint;
Paramètres	hKey: HKEY; lpValueName: Pchar ;
Description	Supprime une valeur d'une clé du registre

Fonction	**RegEnumKeyEx**
Syntaxe	function RegEnumKeyEx(hKey: HKEY; dwIndex: DWORD; lpName: PChar; var lpcbName: DWORD; lpReserved: Pointer; lpClass: PChar; lpcbClass: PDWORD; lpftLastWriteTime: PFileTime): Longint;
Paramètres	hKey: HKEY; dwIndex: DWORD; lpName: PChar; var lpcbName: DWORD; lpReserved: Pointer; lpClass: PChar; lpcbClass: PDWORD; lpftLastWriteTime: PfileTime ;
Description	Enumère les sous-clés d'une clé

Fonction	RegEnumValue
Syntaxe	function RegEnumValue(hKey: HKEY; dwIndex: DWORD; lpValueName: PChar; var lpcbValueName: DWORD; lpReserved: Pointer; lpType: PDWORD; lpData: PByte; lpcbData: PDWORD): Longint;
Paramètres	hKey: HKEY; dwIndex: DWORD; lpValueName: PChar; var lpcbValueName: DWORD; lpReserved: Pointer; lpType: PDWORD; lpData: PByte; lpcbData: PDWORD ;
Description	Enumère les valeurs d'une clé

Fonction	RegFlushKey
Syntaxe	function RegFlushKey(hKey: HKEY): Longint;
Paramètres	hKey: HKEY ;
Description	Ecrit immédiatement les modifications de registre

Fonction	RegSetValueEx
Syntaxe	function RegSetValueEx(hKey: HKEY; lpValueName: PChar; Reserved: DWORD; dwType: DWORD; lpData: Pointer; cbData: DWORD): Longint;
Paramètres	hKey: HKEY; lpValueName: PChar; Reserved: DWORD; dwType: DWORD; lpData: Pointer; cbData: DWORD ;
Description	Attribue une valeur à une clé

FICHIERS D'INITIALISATION

Fonction	GetPrivateProfileInt
Syntaxe	function GetPrivateProfileInt(lpAppName, lpKeyName: PChar; nDefault: Integer; lpFileName: PChar): UINT;
Paramètres	lpAppName, lpKeyName: PChar; nDefault: Integer; lpFileName: Pchar ;
Description	Récupère une valeur entière de clé dans un fichier d'initialisation privé

Fonction	**GetPrivateProfileSection**
Syntaxe	function GetPrivateProfileSection(lpAppName: PChar; lpReturnedString: PChar; nSize: DWORD; lpFileName: PChar): DWORD;
Paramètres	lpAppName: PChar; lpReturnedString: PChar; nSize: DWORD; lpFileName: Pchar ;
Description	Récupère toutes les clés et valeurs d'une section d'un fichier INI privé

Fonction	**GetPrivateProfileSectionNames**
Syntaxe	function GetPrivateProfileSectionNames(lpszReturnBuffer: PChar; nSize: DWORD; lpFileName: PChar): DWORD;
Paramètres	lpszReturnBuffer: PChar; nSize: DWORD; lpFileName: Pchar ;
Description	Récupère tous les noms de segment d'un fichier INI privé

Fonction	**GetPrivateProfileString**
Syntaxe	function GetPrivateProfileString(lpAppName, lpKeyName, lpDefault: PChar; lpReturnedString: PChar; nSize: DWORD; lpFileName: PChar): DWORD;
Paramètres	lpAppName, lpKeyName, lpDefault: PChar; lpReturnedString: PChar; nSize: DWORD; lpFileName: Pchar ;
Description	Récupère une valeur de chaîne pour une clé

Fonction	**WritePrivateProfileSection**
Syntaxe	function WritePrivateProfileSection(lpAppName, lpString, lpFileName: PChar): BOOL;
Paramètres	lpAppName, lpString, lpFileName: Pchar ;
Description	Stocke un segment de fichier avec les clés et valeurs données

Fonction	**WritePrivateProfileString**
Syntaxe	function WritePrivateProfileString(lpAppName, lpKeyName, lpString, lpFileName: PChar): BOOL;
Paramètres	lpAppName, lpKeyName, lpString, lpFileName: Pchar ;
Description	Stocke une valeur de chaîne d'une clé dans un fichier INI privé

INFORMATIONS SUR LE SYSTÈME

Fonction	**GetComputerName**
Syntaxe	function GetComputerName(lpBuffer: PChar; var nSize: DWORD): BOOL;
Paramètres	lpBuffer: PChar; var nSize: DWORD ;
Description	Récupère le nom de l'ordinateur

Fonction	**GetDiskFreeSpace**
Syntaxe	function GetDiskFreeSpace(lpRootPathName: PChar; var lpSectorsPerCluster, lpBytesPerSector, lpNumberOfFreeClusters, lpTotalNumberOfClusters: DWORD): BOOL;
Paramètres	lpRootPathName: PChar; var lpSectorsPerCluster, lpBytesPerSector, lpNumberOfFreeClusters, lpTotalNumberOfClusters: DWORD ;
Description	Récupère les paramètres du disque pouvant être utilisés pour calculer l'espace disponible

Fonction	**GetDriveType**
Syntaxe	function GetDriveType(lpRootPathName: PChar): UINT;
Paramètres	lpRootPathName: Pchar ;
Description	Communique le type de support physique associé à un nom d'accès logique

Fonction	**GetEnvironmentStrings**
Syntaxe	function GetEnvironmentStrings: PChar;
Paramètres	
Description	Communique un pointeur à une liste de chaîne de l'environnement

Fonction	**GetEnvironmentVariable**
Syntaxe	function GetEnvironmentVariable(lpName: PChar; lpBuffer: PChar; nSize: DWORD): DWORD;
Paramètres	lpName: PChar; lpBuffer: PChar; nSize: DWORD ;
Description	Récupère la valeur d'une variable d'environnement

Fonction	**GetLocalTime**
Syntaxe	procedure GetLocalTime(var lpSystemTime: TSystemTime);
Paramètres	var lpSystemTime: TsystemTime ;
Description	Récupère l'heure locale

Fonction	**GetSystemDirectory**
Syntaxe	function GetSystemDirectory(lpBuffer: PChar; uSize: UINT): UINT;
Paramètres	lpBuffer: PChar; uSize: UINT ;
Description	Récupère le nom du répertoire système de Windows

Fonction	GetSystemInfo
Syntaxe	procedure GetSystemInfo(var lpSystemInfo: TSystemInfo);
Paramètres	var lpSystemInfo: TsystemInfo ;
Description	Récupère les informations sur le matériel

Fonction	GetVolumeInformation
Syntaxe	function GetVolumeInformationA(lpRootPathName: PAnsiChar; lpVolumeNameBuffer: PAnsiChar; nVolumeNameSize: DWORD; lpVolumeSerialNumber: PDWORD; var lpMaximumComponentLength, lpFileSystemFlags: DWORD; lpFileSystemNameBuffer: PAnsiChar; nFileSystemNameSize: DWORD): BOOL;
Paramètres	lpRootPathName: PAnsiChar; lpVolumeNameBuffer: PAnsiChar; nVolumeNameSize: DWORD; lpVolumeSerialNumber: PDWORD; var lpMaximumComponentLength, lpFileSystemFlags: DWORD; lpFileSystemNameBuffer: PAnsiChar; nFileSystemNameSize: DWORD ;
Description	Récupère les informations sur le volume et le système de fichiers installé

Fonction	GetWindowsDirectory
Syntaxe	function GetWindowsDirectory(lpBuffer: PChar; uSize: UINT): UINT;
Paramètres	lpBuffer: PChar; uSize: UINT ;
Description	Communique le nom du répertoire de Windows

MANIPULATIONS DE CHAÎNES ET JEUX DE CARACTÈRES

Fonction	CharLower
Syntaxe	function CharLower(lpsz: Pchar): PChar;
Paramètres	lpsz: Pchar ;
Description	Convertit une chaîne en minuscule

Fonction	CharUpper
Syntaxe	function CharUpper(lpsz: Pchar): PChar;
Paramètres	lpsz: Pchar ;
Description	Convertit une chaîne en majuscule

Fonction	IsCharAlpha
Syntaxe	function IsCharAlpha(ch: Char): BOOL;
Paramètres	ch: Char ;
Description	Contrôle si un caractère est alphabétique

Fonction	IsCharAlphaNumeric
Syntaxe	function IsCharAlphaNumeric(ch: Char): BOOL;
Paramètres	ch: Char ;
Description	Contrôle si un caractère est alphanumérique

HORLOGES

Fonction	GetCurrentTime
Syntaxe	function GetCurrentTime: Longint;
Paramètres	
Description	Communique le nombre de tops d'horloge système depuis le démarrage de Windows

Fonction	GetTickCount
Syntaxe	function GetTickCount: DWORD;
Paramètres	
Description	Communique le nombre de tops d'horloge système depuis le démarrage de Windows

GESTION DES SYSTÈMES DE FICHIERS

Cette section est consacrée aux différentes tâches d'entrée-sortie de vos applications. Parmi les méthodes les plus courantes de communication, nous nous intéresserons plus particulièrement aux entrées/sorties de fichiers, y compris un certain nombre de techniques permettant de transmettre, stocker et récupérer des informations sur des fichiers disque. Vous découvrirez également comment imprimer vos œuvres.

ENTRÉE/SORTIE DE FICHIERS

Une des tâches les plus courantes et les plus précieuses en programmation est la manipulation de fichiers. Un fichier n'est rien d'autre qu'une collection ordonnée de données qui est stockée sur un disque dur, une disquette, un CD-ROM, une bande ou tout autre support de masse. De façon évidente, les applications de bases de données doivent pouvoir créer, lire et écrire des fichiers, mais ce n'est pas le seul cas où les fichiers s'avèrent indispensables. On peut ainsi utiliser des fichiers pour stocker les informations de configuration d'une application. Ils permettent de stocker temporairement des informations pour permettre à un programme d'occuper de la mémoire système précieuse à d'autres tâches, puis de recharger les informations dans la mémoire si nécessaire. Les fichiers peuvent même être utilisés pour transmettre des informations d'un programme à un autre. Enfin, bien sûr, les fichiers sont fréquemment utilisés pour enregistrer notre travail dans un traitement de textes ou un tableur. Delphi gère parfaitement les fichiers et de manière très aisée à assimiler. Le but de ce chapitre est de vous montrer comment travailler avec les divers types de fichiers et de vous présenter les fonctions et procédures qui simplifient les tâches liées aux fichiers.

Nous allons parler des attributs et des types de fichier. Vous apprendrez à travailler avec des fichiers texte, des fichiers de type binaires et des fichiers non typés. Vous pourrez également découvrir certaines fonctions liées aux fichiers et répertoires, fonctions qui calquent leur comportement sur des commandes DOS classiques telles que MkDir. Pour terminer, nous traiterons des noms de fichier longs.

ATTRIBUTS DE FICHIERS

Les fichiers ont des attributs spéciaux. Un attribut est une propriété exhibée par un fichier et qui dépend de son paramétrage. Par exemple, si la propriété lecture seule d'un fichier est activée, ce fichier peut être lu mais non mis à jour ni supprimé par la plupart des commandes ou programme DOS. Chaque fichier comporte un octet d'attribut qui stocke les paramètres d'attributs. Chaque paramètre est stocké dans un des huit bits qui composent l'octet. Seuls six bits sont utilisés et, sur ces six bits utilisés, deux servent pour les répertoires et les labels de volume, ne laissant donc que quatre bits pour les attributs eux-mêmes.

Ces quatre attributs sont les suivants :

Attributs de fichier	Constante DELPHI	Description
Fichier en lecture seule	FaReadOnly	Permet qu'on lise le fichier, mais pas qu'on le mette à jour ou qu'on le supprime
Fichier caché	FaHidden	Empêche que le fichier apparaisse dans une liste de répertoires normale
Fichier système	FaSysFile	Marque qu'un fichier est utilisé par le système et empêche qu'il soit visible dans une liste de répertoires normale
Fichier archive	FaArchive	Cet attribut est désactivé si un fichier a été sauvegardé
ID volume	FaVolumeID	Cet attribut sert à créer un ID de volume
Répertoire	FaDirectory	Cet attribut permet d'identifier les répertoires

Ces paramètres d'attributs sont activés ou désactivés par les programmes qui utilisent ou créent les fichiers. Par défaut, à l'exception du bit archive (bit 5), tous les attributs sont désactivés lors de la création. En Delphi, comme dans d'autres langages, vous pouvez modifier ces attributs avant ou après avoir travaillé sur les fichiers. Delphi propose des fonctions telles que FileGetAttr et FileSetAttr qui permettent de lire et de modifier les attributs à loisir. Ce peut être nécessaire si, par exemple, vous avez besoin de manipuler un fichier qui est en lecture seule. Il suffit alors de désactiver l'attribut lecture seule, de travailler sur le fichier puis, les modifications terminées, de réactiver l'attribut lecture seule. Il convient aussi de remarquer que l'on peut activer ou désactiver n'importe quelle combinaison de ces attributs. Comment tout cela fonctionne-t-il ? Découpons notre octet d'attributs en ses 8 composants binaires et intéressons-nous aux bits.

Les bits de l'octet attribut

Bit	Attribut stocké
Bit 0	Lecture seule
Bit 1	Fichier caché
Bit 2	Fichier système
Bit 3	ID volume
Bit 4	Sous - répertoire
Bit 5	Archive
Bit 6	Inutilisé
Bit 7	Inutilisé

Les attributs d'un fichier tout juste créé seraient les suivants :

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0	0	1	0	0	0	0	0

Soit 00100000 en binaire ou 20 en hexa.

Supposons maintenant que vous souhaitiez activer la Lecture seule. Vous pouvez utiliser la fonction FileSetAttr de Delphi avec la constante faReadOnly. Cette constante est égale à 00000001 en binaire ou 01 en hexa. Lorsqu'on a fourni le nom de fichier et la valeur de faReadOnly à FileSetAttr, un OU est effectué sur la valeur et l'octet d'attribut. Dans cet exemple on désactiverait le bit d'archive, ce qui est souhaitable ou non selon ce que vous cherchez à faire. Nous verrons comment travailler avec plusieurs octets par la suite.

En examinant $AD — ou 10101101 en binaire (on lit de droite à gauche ce nombre binaire) —, vous verrez, en vous appuyant sur la table de correspondance des bits de l'octet d'attribut, qu'il correspond à un fichier dont les bits suivants sont activés : Lecture seule, Système, ID de Volume et Archive. Le bit 7 est également activé, mais il ne sert à rien. Ce n'est pas là un octet d'attribut valide. En effet, l'ID de volume et le Sous - répertoire sont des attributs que vous n'avez pas à manipuler car ils sont activés lorsque vous créez un label de volume ou un sous - répertoire. Ils n'ont pas besoin d'être modifiés par le programmeur ou l'utilisateur. Un nombre plus réaliste serait 100001 (binaire) ou $21. En consultant la même table, vous verrez que ce nombre correspond à un fichier dont les attributs Lecture seule et Archive sont activés (le fichier est en lecture seule et le fichier n'a pas été sauvegardé). Intéressons-nous maintenant à la manière dont on active ces bits. Imaginons que vous souhaitiez n'activer que le bit Lecture seule d'un fichier appelé TOTO.TXT. Vous pouvez alors placer la ligne de code suivante dans votre application :

FileSetAttr('TOTO.TXT',faReadOnly);

Dans la déclaration FileSetAttr, la constante faReadOnly est égale au binaire 00000001 ou $01. Cette valeur est copiée dans l'octet d'attribut. Dans notre exemple, le bit Lecture seul passe à 1 et tous les autres à 0.

Si vous souhaitez activer les bits Lecture seule et Archive, il vous suffit d'utiliser la déclaration :

FileSetAttr('TOTO.TXT',faReadOnly+faArchive);

Ceci crée un masque de 00100001.

Maintenant que deux bits sont activés, comment faire pour en désactiver un seul ? Il suffit d'appeler FileSetAttr de nouveau et de ne lui passer qu'un masque contenant les bits que vous souhaitez laisser activés, les autres seront alors désactivés. La ligne qui suit désactive le bit d'archives tout en conservant le bit de Lecture seule :

```
FileSetAttr('TOTO.TXT',faReadOnly);
```

Si vous utilisez cet exemple, tous les bits seront remis à zéro, excepté le bit de Lecture seule. Ce n'est pas forcément ce que vous souhaitez. Ceci s'explique par le fait que la constante faReadOnly qui n'a qu'un bit d'activé est copiée dans l'octet d'attribut. La meilleure manière pour résoudre notre problème consiste à commencer par prendre les attributs d'un fichier pour les stocker dans une variable. Vous pouvez alors créer un masque, et il suffira d'effectuer un OU entre le masque et la variable (contenant les paramètres initiaux) pour ne modifier que les bits désirés.

Le code pour parvenir à ce résultat est le suivant :

```
Var
        FileAttr Integer ;
Begin
        {on récupère le contenu de l'octet Attribut pour MYFILE.TXT}
        FileAttr:=FileGetAttr('MYFILE.TXT');

        {on effectue un OR entre le masque (faReadOnly) et la valeur de l'octet
        d'attribut stockée dans FileAttr, et on place le résultat dans FileAttr}
        FileAttr:=FileAttr OR faReadOnly;

        {On donne pour valeur à l'octet d'attribut la nouvelle valeur stockée dans
        FileAttr}
        FileSetAttr('MYFILE.TXT',FileAttr);
end;
```

TYPES DE FICHIER

Il y a deux types fondamentaux de fichiers : *texte* et *binaire*. Vous pouvez stocker ou formater des données dans ces deux types de fichiers de bien des manières différentes. Comme tout, ces types ont leurs avantages et leurs inconvénients. Ce qui convient dans une situation précise ne convient pas forcément dans une autre. Regardons ensemble les différents types de fichiers et quelques exemples de leurs utilisations. Ce ne sont que des exemples. Vous êtes entièrement libre de formater et de manipuler des données stockées dans un fichier de la manière qui vous convient.

FICHIERS TEXTE

Nous connaissons tous les fichiers texte. Les fichiers texte *sont des fichiers simples contenant des caractères ASCII bruts. Dans un fichier texte, les données sont généralement stockées et récupérées de manière séquentielle, une ligne à la fois. Chaque ligne se termine par des caractères retour chariot (*$D*) et saut de ligne (*$A*). Comme les données sont traitées séquentiellement, une recherche sur un grand fichier texte ou l'apport de nombreuses modifications à un fichier texte peuvent se révéler des tâches fastidieuses et terriblement inefficaces. De manière générale, si vous comptez manipuler des données séquentiellement et que vous n'avez pas besoin d'effectuer des sauts d'un emplacement à un autre dans un même fichier, un fichier texte est parfaitement adapté à vos besoins. Nous allons examiner certaines des fonctions et procédures qui vous permettent de manipuler des fichiers texte, puis nous écrirons un programme qui stocke et lit des données dans un fichier texte.*

La première chose à considérer est le type de variable TextFile. TextFile vous permet de déclarer une variable qui permettra d'identifier le type de fichier que vous allez manipuler. Votre déclaration pourrait être la suivante :

```
Var
    MyFile : TextFile;
```

Maintenant que vous avez une variable du type TextFile, vous devez trouver un moyen de passer ces informations à Delphi ces informations ainsi que le nom du fichier à manipuler. Pour ce faire, vous utilisez la procédure AssignFile. Si vous connaissez déjà Pascal, la procédure Assign doit vous être familière. Delphi a une compatibilité ascendante compatible avec la procédure Assign, mais vous devez utiliser AssignFile en Delphi pour éviter des conflits d'étendue.

On utilise AssignFile de la manière suivante :

```
AssignFile(MonFichier,NomDeFichier)
```

Où MonFichier est la variable que vous avez définie comme un fichier texte et NomDeFichier est une chaîne contenant le nom de fichier que vous souhaitez manipuler. Une fois que vous avez utilisé AssignFile, vous pouvez vous référer au fichier en employant MonFichier. Vous devez aussi connaître certaines procédures permettant d'écrire dans des fichiers texte.

Procédure	Description
ReWrite	Crée et ouvre un nouveau fichier. Tout fichier existant portant le meme nom sera écrasé
WriteLn	Ecrit une ligne de texte dans le fichier ouvert, en ajoutant en bout de ligne une combinaison CR/LF (retour chariot / saut de ligne)
CloseFile	Finit la mise à jour du fichier courant et referme ce dernier

Pour lire des fichiers texte, vous aurez besoin des procédures suivantes :

Procédure	Description
Reset	Ouvre un fichier existant. Les fichiers texte sont ouverts en lecture seule
Readln	Lit la ligne de texte courant dans un fichier de texte ouvert. Chaque ligne s'achève par une combinaison CR/LF.

FICHIERS BINAIRES

Le deuxième type de fichier est le fichier binaire*. Tous les fichiers de type non texte entrent dans cette catégorie. Un fichier binaire n'est rien d'autre qu'un fichier contenant des informations binaires écrites par un programme. Dans le cas des caractères ASCII, le code ASCII représente les informations binaires écrites dans le fichier. A la différence des textes fichier, tout fichier ouvert comme fichier binaire y compris les textes, fichiers de programme, bitmap, etc., peut être lu par votre programme. Dans ce mode, c'est à vous qu'il incombe de déterminer la façon de traiter les données que vous pouvez lire.*

Il existe deux catégories de fichiers binaires : les fichiers typés et les fichiers non typés. Ces catégories sont décrites dans les sections suivantes.

FICHIERS TYPÉS

Le *fichier typé* est l'un des différents types de fichiers binaires. Ce sont des fichiers dont vous avez choisi le format ou la structure, ainsi que le type (et la longueur) des données que vous y stockez, que ce soit des entiers, des réels, des chaînes, etc.

Examinons les procédures et fonctions qui permettent de manipuler les fichiers typés :

AssignFile
Syntaxe : procedure AssignFile(var F : File, Chemin : String);
Utilité : Permet d'affecter un nom de fichier à une variable de fichier à l'attention d'autres fonctions d'E/S de fichier.

Reset
Syntaxe : procedure Reset(var F : File[; RecSize: Word]);
Utilité : Permet d'ouvrir un fichier existant qui a été affecté à une variable de fichier au moyen de AssignFile.

Rewrite
Syntaxe : procedure Rewrite (var F : File[; RecSize: Word]);
Utilité : Permet de créer et d'ouvrir un fichier existant qui a été affecté à une variable de fichier au moyen de AssignFile.

Seek
Syntaxe : procedure Seek (var F : File; N : Longint);
Utilité : Permet de déplacer le pointeur de fichier jusqu'à l'enregistrement spécifié (par N) dans le fichier ouvert.

Read
Syntaxe : procedure Read(F , V1 [, V2,...,Vn]);
Utilité : Permet de lire des enregistrements dans un fichier

Write
Syntaxe : procedure Write (F , V1 [, V2,...,Vn]);
Utilité : Permet d'écrire des enregistrements dans un fichier.

Eof
Syntaxe : function Eof(var F): Boolean;
Utilité : Permet de déterminer si le programme a atteint la fin du fichier. Utilisé en conjonction avec Read.

CloseFile
Syntaxe : procedure CloseFile(var F);
Utilité : Permet de mettre à jour les modifications du fichier et de refermer ce dernier.

FICHIERS NON TYPÉS

Les *fichiers non typés* vous permettent de manipuler les fichiers avec plus de souplesse. Vous pouvez aller en n'importe quel emplacement du fichier, modifier un octet ou un bloc entier, enregistrer les données et fermer le fichier. Lorsque vous écrivez votre code, vous n'avez pas à vous conformer à des structures rigides et votre code peut traiter n'importe quel type de fichiers, de n'importe quelle façon. Il y a un inconvénient cependant. En effet, vous devez écrire votre code en déterminant très exactement à quel endroit du fichier vous souhaitez travailler.

Vous devez pour cela utiliser des pointeurs de fichier, vous appuyer sur une bonne connaissance du fichier lui-même et mettre en œuvre des algorithmes pointus dans votre application. Les tailles des enregistrements peuvent varier, et il incombe au programmeur de déterminer où se trouve un enregistrement de données, et quelle est sa taille. La plus grande prudence est de mise lorsque vous travaillez avec des fichiers non typés, mais le jeu en vaut parfois la chandelle.

Imaginez que vous avez un fichier de taille conséquente dans lequel vous souhaitez remplacer tous les espaces par des virgules. Vous pouvez écrire un programme Delphi simple qui transférera le fichier dans un tampon, bloc par bloc, puis cherchera dans le tampon les espaces et les transformera en virgules au fur et à mesure avant d'enregistrer les modifications sur le disque.

Il est indifférent que le fichier soit un fichier texte ou binaire, de même que la façon dont sont stockées les données dans ce fichier n'importe pas. Pour tout dire, vous êtes totalement libre dans le choix de votre méthode d'accès ou de manipulation des données dans un fichier non typé. Avant de poursuivre sur ce sujet, examinons quelques procédures et fonctions qui vous seront utiles.

BlockRead
Syntaxe : BlockRead(var F: File; var Buf; Count: Word [; var Result: Word]);
Utilité : Lit un bloc de données sur le disque et le place dans un tampon.

BlockWrite
Syntaxe : BlockWrite(var f: File; var Buf; Count: Word [; var Result: Word]);
Utilité : Permet d'écrire un bloc de données de la mémoire vers le disque.

FilePos
Syntaxe : FilePos(var F): Longint;
Utilité : Permet de récupérer la position courante du pointeur de fichier.

GESTION DE FICHIERS, RÉPERTOIRES

La liste des fonctions et procédures d'E/S et de gestion de fichier à connaître n'est pas close. Cet ouvrage ne suffirait pas pour toutes les présenter. Tâchez de vous familiariser aux fonctions et procédures qui figurent dans l'aide en ligne dans les trois rubriques suivantes :

- Routines de Gestion de fichier.
- Routines d'E/S.
- Routines de fichiers texte.

Vous avez déjà eu l'occasion d'utiliser bon nombre des fonctions et procédures qui figurent dans ces rubriques, mais d'autres doivent vous être inconnues. La plupart des fonctions effectuent les mêmes types d'opérations mais de différentes manières.

Ainsi :

```
Var
        MyFileVar : File ;
Begin
        AssignFile(MyFileVar,filename);
        ReWrite(MyFile);
End;
```

Cette routine crée une variable Fichier et lui affecte un nom de fichier. La variable Fichier est utilisée par la procédure Rewrite pour créer et ouvrir le fichier.

Vous pouvez parvenir au même résultat en utilisant la fonction FileCreate, comme le montre le code ci-après :

```
Var
        Handle : Integer;
Begin
        Handle :=FileCreate('Filename');
End;
```

La fonction FileCreate *renvoie un* handle *de fichier si l'opération est réussie. Un handle de fichier n'est rien d'autre qu'une valeur entière qui permettra d'identifier le fichier jusqu'à sa fermeture. Si plus d'un fichier est ouvert, à chaque fichier sera affecté un handle qui lui est propre. Cet handle de fichier est utilisé par de nombreuses fonctions et procédures pour lire, écrire, déplacer le pointeur de fichier, etc., comme le faisaient les procédures et fonctions que nous avons vues, mais qui, elles, utilisaient des variables de fichier.*

Utiliser des handles plutôt que des variables de fichier présente certains avantages. Ainsi, la fonction FileOpen vous permet d'effectuer un OR sur un ensemble de constantes pour définir le mode dans lequel le fichier sera ouvert.

Les constantes de mode de fichier sont indiquées ci-après :

fmOpenRead
fmOpenWrite
fmOpenReadWrite
fmShareCompat
fmShareExclusive
fmShareDenyWrite
fmShareDenyRead
fmShareDenyNone

Pour plus de détails sur ces constantes, vous pouvez vous reporter à l'aide en ligne. Vous trouverez leur description dans l'aide consacrée à l'unité SysUtils. Si vous souhaitez ouvrir un fichier avec un accès exclusif et empêcher qu'un autre utilisateur ou programme puisse y accéder, il vous suffit d'utiliser la constante fmShareExclusive.

Par exemple :

```
MyHandle:=FileOpen(fname,fmShareExclusive);
```

N'oubliez pas que vous pouvez effectuer un OR sur ces constantes pour définir à votre guise les modalités d'accès au fichier. Il peut devenir nécessaire de le faire si vous partagez des fichiers avec d'autres programmes dans le même système, ou avec d'autres utilisateurs dans un réseau.

Dans le tableau ci-après figurent d'autres fonctions et routines dont vous devez connaître l'existence. Prenez le temps de les regarder et de consulter l'aide en ligne à leur sujet.

Erase	Supprime un fichier
FileSize	Donne la taille du fichier spécifié
GetDir	Donne le répertoire courant du lecteur spécifié
MkDir	Crée un sous – répertoire
Rename	Renomme un fichier
RmDir	Supprime un sous – répertoire

De nombreuses fonctions et procédures ayant trait à la manipulation des fichiers n'ont pas pu être abordées ici. Si vous avez lu l'aide en ligne concernant les trois rubriques citées plus haut, vous avez pu compléter vos connaissances.

NOMS LONGS DE FICHIERS

Nous en arrivons enfin aux noms longs ! Vous n'êtes plus limité à des noms de fichiers de 11 caractères (8 pour le nom de fichier, 3 pour l'extension). Sous Windows 95 et NT, les noms de fichiers peuvent occuper jusqu'à 255 caractères, y compris le zéro terminal. La taille maximum d'un chemin est de 260 caractères, y compris le zéro terminal.

Pour des raisons de compatibilité ascendante, un nom court est également créé à partir des six premiers caractères du nom long. Si plusieurs noms longs ont en commun les six premiers caractères, le système d'exploitation utilise un algorithme incrémental de nommage des fichiers en format 8.3. Voici quelques exemples de noms longs avec leur équivalent de nom court :

Nom long	**Nom court**
LongFichierNumero1.TXT	LONGFI~1.TXT
LongFichierNumero2.TXT	LONGFI~2.TXT

Delphi 3 sait tirer parti des noms longs. Vous n'avez rien de particulier à faire, Delphi et le système d'exploitation gèrent tout ceci de façon transparente pour vous. Il vous suffit d'utiliser le nom long dans vos fonctions, procédures et programmes. Vous pouvez également continuer d'utiliser l'ancien format 8.3, la compatibilité est assurée.

AUTRES PRIMITIVES UTILES

Même en environnement Windows, un programme peut être lancé avec des paramètres optionnels sur sa ligne de commande (on peut automatiser ce lancement paramétré en spécifiant les paramètres dans le menu 'Propriétés ' du Gestionnaire de programmes après avoir sélectionné l'icône du programme.

Pour récupérer les paramètres de la ligne de commande il faut utiliser les procédures :

Function ParamCount : Word ;

ParamCount renvoie le nombre de paramètres passés dans la ligne de commande.

function ParamCount: Integer;

Description

La fonction ParamCount renvoie le nombre de paramètres passés au programme par la ligne de commande. Les paramètres sont séparés avec des espaces ou des tabulations. Utilisez les guillemets pour rassembler plusieurs mots en un seul paramètre (comme des noms de fichier long contenant des espaces).

Exemple :

```
var
        I: Word;
        Y: Integer;
begin
        Y := 10;
        for I := 1 to ParamCount do
        begin
                Canvas.TextOut(5, Y, ParamStr(I));
                Y := Y + Canvas.TextHeight(ParamStr(I)) + 5;
        end;
end;
```

Function ParamStr (Index) : String ;

ParamStr renvoie le paramètre spécifié depuis la ligne de commande.

function ParamStr(Index: Integer): string;

Description

Index est une expression de type Integer. La fonction ParamStr renvoie, soit le paramètre de la ligne de commande correspondant à la position Index, soit une chaîne vide si Index est supérieur à ParamCount. Par exemple, une valeur Index de 2 renvoie le deuxième paramètre de la ligne de commande.

ParamStr(0) renvoie le chemin et le nom de fichier du programme en cours d'exécution (par exemple, C:\TEST\MYPROG.EXE).

Exemple

```
var
        I: Word;
        Y: Integer;
begin
        Y := 10;
        for I := 1 to ParamCount do
        begin
                Canvas.TextOut(5, Y, ParamStr(I));
                Y := Y + Canvas.TextHeight(ParamStr(I)) + 5;
        end;
end;
```

Function ChangeFileExt (const Fichier, Extension: string) :string ;

ChangeFileExt change l'extension d'un fichier.

```
function ChangeFileExt(const FileName, Extension: string): string;
```

Description

La fonction ChangeFileExt prend le nom de fichier transmis par FileName et modifie l'extension du fichier par celle transmise par Extension. ChangeFileExt ne renomme pas le fichier lui-même, elle crée simplement une chaîne contenant le nouveau nom du fichier.

Exemple :

```
function INIFileName: string;
begin
        INIFileName := ChangeFileExt(ParamStr(0), '.INI');
end;
```

Function DeleteFile (const Fichier: string) : Boolean ;

DeleteFile supprime un fichier du disque et renvoie False si elle échoue.

```
function DeleteFile(const FileName: string): Boolean;
```

Description

La fonction DeleteFile efface sur le disque le fichier intitulé FileName. Si le fichier ne peut être supprimé ou n'existe pas, la fonction renvoie False mais ne provoque pas d'exception.

Exemple :

```
if FileExists(FileName) then
        if MsgBox( 'Voulez-vous vraiment supprimer ' +
                        ExtractFileName(FileName) + '?'), []) = IDYes then
        DeleteFile(FileName);
```

Function ExpandFileName (const Fichier : string) : string ;

ExpandFileName renvoie le chemin complet de Filename.

function ExpandFileName(const FileName: string): string;

Description

La fonction ExpandFileName renvoie une chaîne contenant le chemin d'accès développé du fichier transmis par le paramètre FileName. Le chemin développé ainsi renvoyé comprend la lettre du lecteur de disque ainsi que le répertoire et les sous-répertoires éventuels suivis du nom de fichier et de son extension.

Exemple :

Le code suivant convertit un nom de fichier en nom complet :
MyFileName := ExpandFileName(MyFileName);

Function ExtractFileExt (const Fichier : string) : string

ExtractFileExt renvoie la partie extension de FileName.

function ExtractFileExt(const FileName: string): string;

Description

La fonction ExtractFileExt extrait l'extension d'un nom de fichier donné. La chaîne renvoyée comprend le point séparant le nom et l'extension. Cette chaîne est vide si le nom du fichier ne comprend pas d'extension. Le code suivant renvoie l'extension d'un nom de fichier :

MyFilesExtension := ExtractFileExt(MyFileName);

Function ExtractFilePath (const Fichier: string) : string ;

La fonction ExtractFilePath extrait le lecteur et le répertoire d'un nom de fichier.

function ExtractFilePath(const FileName: string): string;

Déclaration

La chaîne renvoyée est composée des caractères de gauche de FileName, jusqu'aux deux points ou la barre oblique inverse qui séparent le chemin du nom et de l'extension. La chaîne renvoyée est vide si FileName ne contient pas de partie lecteur et répertoire.

Exemple :

L'exemple suivant crée un alias qui pointe sur le répertoire de l'application. Remarquez que cet alias est automatiquement supprimé lorsque l'application est terminée et qu'il n'est pas sauvegardé dans le fichier de configuration du BDE.

```
begin
        with Session do
        begin
                ConfigMode := cmSession;
                try
                        AddStandardAlias('TEMPDB', ExtractFilePath(ParamStr(0)),
                        'PARADOX');
                finally
                        ConfigMode := cmAll;
                end;
end;
```

Function FileAge (const Fichier : string) : Longint ;

FileAge renvoie la date et l'heure du fichier spécifié.

```
function FileAge(const FileName: string): Integer;
```

Description

La fonction FileAge renvoie l'âge du fichier FileName sous la forme d'un Integer. La valeur renvoyée est -1 si le fichier n'existe pas.

Function FileCreate (const Fichier: string) : Integer;

FileCreate crée un nouveau fichier.

```
function FileCreate(const FileName: string): Integer;
```

Description

La fonction FileCreate crée un nouveau fichier avec le nom spécifié. Si la valeur renvoyée est positive, la fonction s'est bien déroulée et cette valeur correspond au descripteur du nouveau fichier. Si la valeur renvoyée vaut -1, cela indique qu'une erreur s'est produite.

L'utilisation de gestionnaires de variables de fichier Pascal non natif tels que FileCreate est déconseillée. Pour plus d'informations sur l'utilisation des routines de gestion de fichier, voir FileOpen.

Function FileOpen (const Fichier: string) : Integer;

FileOpen ouvre un fichier en utilisant le mode d'accès spécifié.

function FileOpen(const FileName: string; Mode: Integer): Integer;

Description

La valeur du mode d'accès résulte d'un OU logique entre une des constantes fmOpenXXXX avec une des constantes fmShareXXXX. Si la valeur renvoyée est positive, la fonction s'est bien déroulée et la valeur représente le descripteur du fichier ouvert. Si la valeur renvoyée vaut -1, cela indique qu'une erreur s'est produite.

Remarque

L'utilisation de gestionnaires de variables de fichiers Pascal non natif tels que FileOpen est déconseillée. Ces routines sont identiques aux fonctions de l'API Windows en ce sens qu'elles renvoient les descripteurs de fichiers et non les variables de fichier Pascal classique. Ces dernières sont des routines d'accès aux fichiers de bas niveau. Pour des opérations sur des fichiers classiques, utilisez les fonctions AssignFile, Rewrite, Reset au lieu de FileOpen.

Function FileExists (const Fichier : string) : Boolean ;

FileExists teste si FileName existe.

function FileExists(const FileName: string): Boolean;

Description

La fonction FileExists renvoie True si le fichier FileName existe. Dans le cas contraire, FileExists renvoie False.

Function FileGetAttr (const Fichier : string) : Integer;

FileGetAttr renvoie les attributs du fichier FileName.

function FileGetAttr(const FileName: string): Integer;

Description

Ces attributs peuvent être récupérés à l'aide de l'opérateur ET (AND) et des constantes définies dans TsearchRec (faXXXXX). Une erreur s'est produite si la valeur renvoyée est -1. Les constantes pouvant être utilisées pour tester la valeur renvoyée sont :

Constante	Valeur	Description
faReadOnly	$01	Fichiers en lecture seule
faHidden	$02	Fichiers cachés
faSysFile	$04	Fichiers système
faVolumeID	$08	Fichiers d'identificateurs de volume

faDirectory	$10	Fichiers répertoire
faArchive	$20	Fichiers archive
faAnyFile	$3F	N'importe quel fichier

On peut combiner les attributs de fichier en ajoutant leurs constantes ou valeurs. Par exemple, pour rechercher des fichiers cachés et en lecture seule en plus des fichiers normaux, on peut utiliser le masque (faReadOnly + faHidden).

Function FileSearch (const Nom, ListRep : string) : string ;

FileSearch recherche un fichier dans le chemin DOS spécifié.

function FileSearch(const Name, DirList: string): string;

Description

La fonction FileSearch recherche dans les répertoires transmis par DirList un fichier intitulé Name. DirList doit respecter le format d'un chemin d'accès DOS : les noms de répertoire doivent être séparés par des points-virgules. Si FileSearch localise un fichier correspondant à Name, elle renvoie une chaîne contenant le chemin d'accès développé à ce fichier. Si aucune correspondance n'est trouvée, FileSearch renvoie une chaîne vide.

Exemple :

FoundIt := FileSearch('FIND.DLL',MyAppDir+'\';'+WinDir+';'+WinDir+'\SYSTEM');

Function FileSetAttr (const Fichier: string ; Attr : Integer) :Integer ;

FileSetAttr définit les attributs du fichier spécifié.

function FileSetAttr(const FileName: string; Attr: Integer): Integer;

Description

La fonction FileSetAttr définit les attributs du fichier FileName à partir de la valeur transmise par Attr. La valeur d'attribut est constituée en faisant appel à l'opérateur OU et aux constantes faXXXX appropriées. La valeur renvoyée est zéro si l'exécution de la fonction réussit. Sinon, cette valeur est un code d'erreur Windows.

Function FileSize (var F): Longint;

FileSize renvoie la taille d'un fichier (en octets) ou le nombre d'enregistrements dans le fichier.

function FileSize(var F): Integer;

Description

La fonction FileSize renvoie la taille en octets du fichier F. Pour utiliser FileSize, le fichier doit être ouvert. F est une variable fichier. Si le fichier est vide, FileSize(F) renvoie 0.

Remarque

FileSize ne peut être utilisée avec un fichier texte.

Exemple :

```
var
        f: file of Byte;
        size : Longint;
        S: string;
        y: integer;
begin
        if OpenDialog1.Execute then
        begin
                AssignFile(f, OpenDialog1.FileName);
                Reset(f);
                size := FileSize(f);
                S := 'Taille du fichier en octets: ' + IntToStr(size);
                y := 10;
                Canvas.TextOut(5, y, S);
                y := y + Canvas.TextHeight(S) + 5;
                S := 'Positionnement au milieu du fichier...';
                Canvas.TextOut(5, y, S);
                y := y + Canvas.TextHeight(S) + 5;
                Seek(f,size div 2);
                S := 'Position actuelle: ' + IntToStr(FilePos(f));
                Canvas.TextOut(5, y, S);
                CloseFile(f);
        end;
end;
```

Function RenameFile(const AncienNom, NouveauNom: string): Boolean;

RenameFile renomme le fichier identifié par OldName.

function RenameFile(const OldName, NewName: string): Boolean;

Description

La fonction RenameFile tente de changer le nom du fichier indiqué de OldFile en NewFile. Si l'opération réussit, RenameFile renvoie True. Si le fichier n'a pas pu être renommé (si, par exemple, un fichier intitulé NewName existe déjà), la fonction renvoie False.
Le code suivant renomme un fichier :

```
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
    ErrorMsg('Erreur de renommage de fichier');
```

IMPRIMER

Bien que nous vivions dans un monde de plus en plus électronique, dans lequel nous utilisons télécopie, messages électroniques et logiciels de présentation, il vient toujours un moment où il est nécessaire d'imprimer du texte ou un graphique venant d'un programme, pour générer des formulaires ou des brochures par exemple.

Nous verrons deux méthodes pour imprimer directement à partir de programmes Delphi, ce qui peut faire gagner beaucoup de temps si vous n'avez pas besoin de toutes les fonctionnalités de QuickReport ou d'autres logiciels d'impression. Nous parlerons des techniques d'impression de base, consistant à envoyer une ligne ou une chaîne de texte à la fois vers l'imprimante, comme c'est le cas dans un programme DOS simple en Pascal. Cette section examine également les objets imprimante disponibles dans Delphi. A l'aide de ces objets, vous pouvez imprimer du texte. Vous verrez comment utiliser les boîtes de dialogue d'Impression et comment imprimer des graphiques. Nous aborderons également toutes les bases vous permettant de procéder à des impressions dans vos applications sans avoir besoin d'utiliser pour cela des produits extérieurs.

BASES DE L'IMPRESSION EN PASCAL

Si vous connaissez déjà les techniques d'impression en Pascal ou dans d'autres langages, vous ne serez pas surpris. L'impression dans sa plus simple expression consiste à créer une variable de fichier et à l'affecter à l'imprimante. Vous utilisez alors une déclaration writeln pour envoyer le texte vers l'imprimante. Ce type d'impression est des plus primitifs comparé aux fonctionnalités dont vous disposez dans Windows, mais il suffit parfois amplement. Imaginez par exemple qu'un ordinateur est connecté à une imprimante texte qui permet d'obtenir des sorties papier des mesures d'un instrument. Ou que vous souhaitez imprimer une liste simple, sans avoir besoin d'utiliser de graphiques, de polices et sans formatage particulier.

Le code ci-après utilise une déclaration writeln pour imprimer :

```
var
        P : TextFile;
begin
        AssignPrn(P);
        rewrite(P);
        writeln(P,'Test d''impression');
        CloseFile(P);
end;
```

Comme vous pouvez le voir, on déclare une variable P de type TextFile. On utilise ici une variante de Assign, AssignPrn. Cette fonction affecte la variable au port de l'imprimante, le traitant comme un fichier. Il faut ensuite ouvrir le port de l'imprimante et on utilise rewrite à cet effet. Le texte est envoyé à l'imprimante par le biais de la procédure writeln et le port de l'imprimante est fermé avec CloseFile. Il est important de fermer le port de l'imprimante pour terminer l'opération. Tout texte restant encore en mémoire est envoyé vers l'imprimante et le port est fermé, tout comme un fichier.

IMPRIMER AVEC L'OBJET TPRINTER DE DELPHI

La place nous manque pour décrire en détail toutes les propriétés et méthodes des objets imprimante. Dans cette partie, nous développerons quelques programmes donnant des exemples d'utilisation des objets imprimante. Vous verrez ainsi comment doter vos applications de fonctionnalités d'impression.

En Delphi, vous utilisez l'objet Tprinter pour accéder à l'interface d'impression Windows. L'unité Printers de Delphi contient la variable Printer qui est déclarée comme instance de l'objet Tprinter :

```
Printer : Tprinter;
```

Pour utiliser l'objet TPrinter, vous devez ajouter l'unité Printers à la clause uses de votre code. A la différence d'autres unités usuelles, Printers n'est pas ajouté d'office par Delphi.

```
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls,Printers;
```

Cela fait, vous pouvez utiliser Printer pour référencer des propriétés dans l'objet TPrinter.

L'OBJET TPRINTER

Avant de pouvoir utiliser l'objet TPrinter, vous devez connaître certaines de ses propriétés et méthodes :

Canvas Déclarée comme une instance de l'objet TCanvas. Le canevas est l'endroit où la page ou le document est construit en mémoire avant d'être imprimé. Ses propriétés Pen et Brush vous permettent d'y dessiner et d'y placer du texte.

TextOut Méthode de l'objet TCanvas qui permet d'envoyer du texte vers le canevas.

BeginDoc Permet de lancer une tâche d'impression.

EndDoc Permet de terminer une tâche d'impression. L'impression proprement dite ne commence pas tant que EndDoc n'a pas été appelé.

PageHeight Provoque un saut de page sur l'imprimante et redéfinit à (0,0) la valeur de la propriété Pen du canevas.

PageNumber Renvoie le numéro de la page actuellement imprimée.

Ainsi, si vous souhaitez imprimer du texte à l'aide de l'objet Printer, vous écrirez quelque chose comme :

```
Printer.BeginDoc;
Printer.Canvas.TextOut(10,10,'J''imprime avec l''objet Printer');
Printer.EndDoc;
```

Ce code provoque l'impression du texte J'imprime avec l'objet Printer à partir du dixième pixel à droite et du dixième pixel en partant du haut du canevas. BeginDoc lance l'impression. Le texte est envoyé au canevas à l'aide de la propriété TextOut du canevas. EndDoc déclenche l'impression du texte proprement dite et termine la tâche d'impression.

Ces propriétés et méthodes ne sont que la partie émergée de l'iceberg Tprint, mais elles sont suffisantes pour créer le programme d'impression de fichier.

COMPOSANTS DE GESTION DE L'IMPRESSION

Vous avez sans doute vu des logiciels du commerce qui utilisent des boîtes de dialogue pour sélectionner des options d'impression telles que le nombre de copies, le rassemblement de copies et l'orientation de la page. Il n'est pas difficile avec Delphi d'obtenir ces fonctions. Il vous suffit de placer le composant TPrinterDialog sur la page et d'ajouter quelques lignes de code. Votre application dispose alors de boîtes de dialogue Impression. Le code qui le permet est le suivant :

```
if PrintDialog1.Execute then
Begin
        {votre code d'impression}
end;
```

Lorsque ce code est exécuté, la boîte de dialogue Impression standard de Windows apparaît, permettant à l'utilisateur de sélectionner des options pour la tâche d'impression en attente. Ces sélections faites, la tâche d'impression est menée à bien en tenant compte des paramètres spécifiés (sauf pour le nombre de copies, pour lequel il est nécessaire de créer une boucle). Ce n'est pas plus compliqué que ça.

IMPRIMER DES GRAPHIQUES

Vous avez imprimé du texte en utilisant la méthode traditionnelle et vous avez également envoyé du texte par le biais de l'objet TPrinter. Mais qu'en est-il des représentations graphiques ? Vous désirez peut-être créer des logos, des graphiques et d'autres informations non textuelles. Dans cette partie, nous verrons comment imprimer pratiquement n'importe quel graphique. Vous n'êtes en fait limité que par l'imprimante que vous utilisez.

Envoyer des graphiques à l'imprimante n'est pas très différent d'envoyer des graphiques à l'écran. Vous utilisez la propriété Canvas de l'objet TPrinter et ses propriétés et méthodes pour dessiner ou disposer des graphiques sur le canevas. Vous pouvez en fait concevoir des graphiques en commençant par les envoyer à l'écran. Lorsque vous aurez une bonne idée de leur aspect, il vous suffira de modifier le code pour envoyer le graphique à l'imprimante.

Prenons un exemple. Le code ci-après trace un cercle dans le coin supérieur gauche de la fiche Form1 :

```
begin
        {Définit une épaisseur de crayon de 5 pixels }
        Form1.Canvas.Pen.Width:=5;

        {Dessine une ellipse dont le coin supérieur gauche est à 0,0 et le coin
        inférieur droit à 200,200}
        Form1.Canvas.Ellipse(0, 0, 200, 200);
end;
```

Le code suivant dessine le même cercle, au même emplacement sur le canevas de TPrinter et l'envoie à l'imprimante :

```
begin
        {début de la tâche d'impression }
        Printer.BeginDoc;

        { Définit une épaisseur de crayon de 5 pixels }
        Printer.Canvas.Pen.Width:=5;

        { Dessine une ellipse dont le coin supérieur gauche est à 0,0 et le coin
        inférieur droit à 200,200}
        Printer.Canvas.Ellipse(0, 0, 200, 200);

        {fin et impression de la tâche d'impression }
        Printer.EndDoc;
end;
```

En ajoutant les lignes BeginDoc et EndDoc, et en transformant Form1 en Printer pour pointer vers le canevas de l'imprimante plutôt que celui de la fiche, vous pouvez envoyer le même graphique à l'imprimante.

GDI ET PROGRAMMATION GRAPHIQUE

Delphi propose de nombreuses fonctions permettant de créer facilement des applications graphiques. Dans ce chapitre, nous vous présenterons la base de la création d'applications graphiques et vous montrerons comment utiliser des techniques graphiques et multimédias sophistiquées.

LE CANVAS

Les programmeurs doivent comprendre comment afficher des images dans des applications ou comment manipuler des points, des formes, des lignes et des couleurs. Windows 95 et Windows NT offrent des fonctionnalités puissantes permettant à des applications graphiques de haut niveau de tirer parti des ressources système le plus efficacement possible. Delphi vous offre une gamme étendue de composants et de méthodes cachant au développeur une grande partie des détails de l'implémentation système. C'est un plus pour celui qui découvre les graphismes car il peut se concentrer sur leur moyen de fonctionnement au lieu de perdre son temps à apprendre la manipulation d'appels complexes au système d'exploitation.

COORDONNÉES

Vous savez sans doute ce que sont des coordonnées. Tous les composants visuels ont une propriété Top (haut) et une propriété Left (gauche). Les valeurs stockées dans ces propriétés déterminent la position du composant sur la fiche. Autrement dit, le composant est placé aux coordonnées X, Y, où X est la propriété Left et Y la propriété Top. Les valeurs en X et Y (ou Left et Top) sont exprimées en pixels. Un pixel est la plus petite zone d'écran manipulable.

PROPRIÉTÉ CANVAS

La propriété Canvas est la zone de dessin sur une fiche et sur d'autres composants graphiques; elle permet au code Delphi de manipuler la zone de dessin en cours d'exécution. L'une de ses principales caractéristiques est qu'elle est constituée de propriétés et de méthodes facilitant la manipulation de graphiques dans Delphi. Toutes les manipulations formelles et les comptabilités diverses sont cachées dans l'implémentation de l'objet Canvas.

La partie suivante présente les fonctions de base vous permettant d'effectuer des opérations graphiques dans Delphi à l'aide de l'objet Canvas.

PIXELS

D'un point de vue conceptuel, toutes les opérations graphiques reviennent à définir la couleur des pixels de la surface de dessin. En Delphi, vous pouvez manipuler les pixels individuellement.

Aux débuts de l'informatique, un pixel était soit activé, soit désactivé, il était donc noir ou blanc (ou vert ou ambre). Les pixels peuvent maintenant prendre une vaste gamme de couleurs. Leur couleur peut être spécifiée soit comme une couleur prédéfinie, telle que clBlue, soit comme un mélange arbitraire des trois couleurs rouge, vert et bleu.

Pour accéder aux pixels d'une fiche, vous devez utiliser la propriété Canvas de la fiche et la propriété Pixels du Canvas. La propriété Pixels est un tableau à deux dimensions correspondant aux couleurs du Canvas.Pixels[10,20] correspond à la couleur du pixel situé à 10 pixels à droite et à 20 pixels en bas de l'origine. Vous traitez le tableau pixel comme toute autre propriété : pour modifier la couleur d'un pixel, affectez-lui une nouvelle valeur ; pour déterminer sa couleur, lisez la valeur.

UTILISER DES PIXELS

Dans l'exemple suivant, nous utilisons la propriété Pixels pour dessiner une courbe de sinus dans la fiche principale. Le seul composant de la fiche est un bouton qui dessine la courbe lorsqu'on clique dessus. Nous utilisons les paramètres Width (largeur) et Height (hauteur) de la fiche afin que la courbe prenne place sur 70 % de la hauteur de la fiche et sur l'ensemble de la longueur.

```
unit unitSine;
interface

uses
        Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
        StdCtrls;

type
        TForm1 = class(TForm)
                DrawSine: TButton;
                procedure DrawSineClick(Sender: TObject);
        private
                { Déclarations privées }
        public
                { Déclarations publiques }
        end;

var
        Form1: TForm1;

implementation
{$R *.DFM}


procedure TForm1.DrawSineClick(Sender: TObject);
var
        X, Y : real;
        PX, PY, HalfHeight : longint;
```

```
begin
        { On détermine la moitié inférieure de la fiche }
        HalfHeight := Form1.Height div 2;
        for PX:=0 to Form1.Width do
        BEGIN
                {On met à l'échelle X en fonction de 2 PI pour décrire une période }
                X := PX * (2*PI/Form1.Width);
                Y := sin(X);
                PY := trunc(0.7 * Y * HalfHeight) + HalfHeight;

                {On rend le pixel noir (0 d'intensité RVB)}
                Canvas.Pixels[PX,PY] := 0;
        END;
end;
end.
```

Dans cette application, on commence par déterminer la hauteur de la fiche afin d'échelonner correctement la sinusoïde. Le compteur PX va ensuite de 0 à la largeur de la fiche, et une valeur Y est calculée en chaque point de la sinusoïde. La coordonnée Y est multipliée par un facteur d'échelle afin que la courbe soit bien placée sur la fiche. Le point est dessiné en définissant la propriété Canvas.Pixel.

Chaque canevas dispose d'un crayon imaginaire lui permettant de tracer des lignes et des formes. Pensez au crayon du canevas comme à un véritable crayon sur une feuille de papier. On peut déplacer ce crayon de deux façons. La première consiste à le déplacer en touchant le papier afin de laisser une marque. La deuxième consiste à le lever, ce qui ne laisse pas de marques. En outre, des attributs lui sont associés. Le crayon possède ainsi une couleur et une épaisseur de trait spécifiques. Pour le déplacer sans dessiner, il suffit d'utiliser la méthode MoveTo.

La ligne de code ci-après déplace le crayon jusqu'aux coordonnées 23,56.

```
Form1.Canvas.MoveTo(23,56);
```

TRACER DES LIGNES

Pour tracer une ligne droite allant de la position actuelle du crayon jusqu'à une autre position, il suffit d'utiliser la méthode LineTo. LineTo n'a besoin que des coordonnées de la destination du crayon, et trace alors une ligne droite de la position actuelle jusqu'à la nouvelle.

La procédure suivante utilise la propriété LineTo pour dessiner un motif original.

```
procedure TForm1.DrawSineClick(Sender: TObject);
var
        X, Y : real;
        PX, PY, Offset, HalfHeight : longint;
```

```
begin
        { On détermine les coordonnées de la moitié inférieure de la fiche }
        HalfHeight := Form1.Height div 2;
        For OffSet := -10 to 10 do
        BEGIN
                PX := 0;
                While PX < Form1.Width do
                BEGIN
                        X := PX * (2*PI/Form1.Width);
                        Y := sin(X);
                        PY := trunc(0.7 * Y * HalfHeight)+HalfHeight + (Offset *10);
                        IF (PX = 0) Then
                                canvas.MoveTo(PX,PY);
                        canvas.LineTO(PX,PY);
                        PY := trunc(0.7*Y*HalfHeight)+HalfHeight+((Offset-1) *10);
                        canvas.LineTO(PX,PY);
                        PX := PX +15;
                END;
        END;
end;
end.
```

Ce programme est presque identique à celui du tracé de courbe sinus, à ceci près qu'ici nous créons des interstices entre les points, que nous connectons ensuite avec des lignes. Lors de la première itération (lorsque PX=0), on utilise la méthode MoveTo *pour aller au premier point. Il nous suffit ensuite d'utiliser* MoveTo *pour nous déplacer vers tous les points suivants.*

DESSINER DES POLYGONES

En plus du tracé de lignes droites, le canevas dispose de méthodes pour tracer des formes. Vous pouvez en tracer certaines, telles que des rectangles, en utilisant des méthodes spécifiques, mais vous pouvez en tracer d'autres en utilisant une série de points. Ces formes sont appelées polygones. Delphi dispose de plusieurs méthodes pour tracer des formes remplies. Voyons pour commencer les polygones contourés (nous verrons par la suite comment remplir des objets graphiques). On compte parmi les polygones usuels les triangles, les octogones ou les trapézoïdes. Pour tracer un polygone, vous passez à la fonction PolyLine une série de points qu'elle interconnecte au moyen de lignes.

Vous passez à la fonction PolyLine un tableau de point (c'est un concept qui doit vous sembler nouveau). Jusqu'ici, tout se faisait au moyen de coordonnées et vous passiez à la fonction LineTo des valeurs en X et Y.

Delphi comporte un type TPoint*, qui encapsule les valeurs en X et Y dans un enregistrement unique appelé* point*. La façon la plus simple de créer un point consiste à utiliser la fonction* Point*.* Point *prend une valeur X et une valeur Y et renvoie en enregistrement* TPoint*. Vous remarquerez que le premier et le dernier points ne sont pas forcément connectés, vous devez donc en spécifier un dernier identique au premier si vous souhaitez obtenir un polygone fermé.*

Ainsi, l'appel au PolyLine suivant trace un rectangle :

```
Form1.Canvas.PolyLine([Point(10,10),Point(100,100),Point(50,75),Point(10,10)]);
```

Vous pouvez étendre ce principe pour créer une procédure traçant un polygone symétrique composé d'un nombre arbitraire de côtés entré par l'utilisateur. Si l'utilisateur entre 8, la procédure trace un octogone ; un 6 donne un hexagone. Il suffit d'utiliser des principes simples de géométrie en plaçant les sommets sur un cercle. Le Listing suivant vous montre la source de ce programme.

```
procedure TForm1.DrawPolyClick(Sender: TObject);
var
        Sides, Count : integer;
        PolyArray : Array[0..15] of TPoint;
begin
        Sides := strtoint(NumSides.Text);
        If Sides > 15 then Sides := 15; /* Le tableau ne contient que 15 points*/
        For Count := 0 to Sides do
        BEGIN
                {On utilise les points d'un cercle. On choisit des sommets comme
                points}
                PolyArray[Count]  :=  Point(TRUNC(SIN((2*PI)*COUNT/Sides)*
                30)+(Form1.Width div 2),TRUNC(COS((2*PI)*COUNT/Sides)* 30)+
                (Form1.Height div 2));
        END;
        {On connecte le dernier point au premier et on définit tous les points restants
        comme égaux au premier point }
        For Count := Sides+1 to 15 do
                PolyArray[Count] := PolyArray[0];

        {On dessine le polygone}
        Form1.Canvas.PolyLine(PolyArray);
end;
```

Cette procédure calcule des points régulièrement espacés sur un cercle. Le nombre de points calculés dépend du nombre de côtés sélectionnés. Les points sont connectés en utilisant la méthode PolyLines *sur le canevas. Pour fermer le polygone, il suffit de définir les points inutilisés comme étant égaux au point de départ.*

MODIFIER LES ATTRIBUTS DE CRAYON

Toutes les formes tracées jusqu'ici utilisaient le crayon par défaut. Il est possible de changer la couleur, l'épaisseur de trait et le style du crayon. En Delphi, vous accédez au crayon par le biais du canevas, qui dispose d'une propriété Pen. Les principales propriétés par le crayon son Color, Width, Style et Mode.

COLOR

Vous pouvez définir la couleur du crayon en utilisant les mêmes méthodes que pour la couleur de la fiche. Ainsi, pour définir comme bleue la couleur du crayon, procédez ainsi :

```
Form1.Canvas.Pen.Color := clBlue;
```

Vous pouvez également utiliser la ligne suivante :

```
Form1.Canvas.Pen.Color := RGB(0,0,255);
```

Pour afficher toutes les nuances de gris, utilisez la procédure suivante :

```
procedure TForm1.DrawGreyClick(Sender: TObject);
var
        Count : Integer;
begin
        For Count := 0 to 255 do
        BEGIN
                Form1.Canvas.Pen.Color := RGB(Count,Count,Count);
                Form1.Canvas.MoveTo(Count,0);
                Form1.Canvas.LineTo(Count,100);
        end;
end;
```

WIDTH ET STYLE

La propriété Width définit la largeur du crayon en pixels. La propriété Style donne au crayon des tracés variés, tels que pointillés ou tirets. Les valeurs valides pour la propriété Style sont psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear et psInsideFrame. Si vous souhaitez un crayon rouge, large de trois pixels et traçant en pointillé, exécutez ce qui suit :

```
Form1.Canvas.Pen.Color := clRed;
Form1.Canvas.Pen.Width := 3;
Form1.Canvas.Pen.Style := psDot;
```

MODE

La propriété Mode du crayon lui permet d'interagir avec son environnement. Un mode pmNot, par exemple, fait tracer le crayon dans une couleur inverse de celle du fond (l'inverse de chaque bit par conséquent). Le crayon utilise le mode pmCopy par défaut, lui faisant utiliser la couleur courante. Vous pouvez déterminer la couleur du crayon en regardant la propriété color.

La propriété Mode vous permet de faire des animations simples. Vous pouvez créer une animation en redessinant une partie d'une image et en montrant les modifications au fur et à mesure. Le cerveau a alors l'illusion du mouvement. Pour des animations simples, vous pouvez utiliser les modes pmXor ou pmNotXor pour dessiner un objet, puis l'enlever sans modifier le fond. Rappelez-vous que chaque pixel stocke une couleur sous forme d'une séquence de bits. Modifier un pixel en effectuant un Ou exclusif (XOR) sur le pixel courant change la couleur. L'opération XOR prend deux opérandes considérés comme des ensembles

de bits et renvoie True si l'un des opérandes est vrai (mais pas les deux). Procéder de même une nouvelle fois remet le pixel dans sa couleur initiale.

Les étapes suivantes vous montrent ce qu'il se passe.

1. Le pixel de fond a une valeur de 0110011.
2. Vous effectuez un Ou exclusif de cette valeur avec 1111000.
3. Le nouveau pixel a une valeur de 1001011.

Lorsque vous souhaitez effacer le pixel et refaire apparaître le fond, il suffit de reprendre le même procédé :

1. Le pixel a actuellement pour valeur 1001011.
2. Vous effectuez un Ou exclusif de cette valeur avec 1111000.
3. Vous obtenez 0110011 comme résultat (la valeur initiale).

L'avantage de cette méthode est que vous n'avez pas besoin de stocker les informations concernant le fond, elles sont retrouvées automatiquement. L'inconvénient est que vous n'obtenez pas tout à fait l'image souhaitée car vous faites figurer les informations de fond dans la couleur du pixel. Dans la procédure suivante, on utilise cette technique pour animer un triangle se déplaçant sur une fiche. Vous pouvez voir une boîte rouge sur la fiche, le triangle bleu passera sur la boîte sans la modifier.

```
procedure TForm1.SimpleAnimateClick(Sender: TObject);
var
        Count : Integer;
        Pause : real ;
begin
        { On dessine une boîte }
        Form1.Canvas.Pen.mode := pmCopy;
        Form1.Canvas.Pen.Color := clRed;
        Form1.Canvas.PolyLine([point(50,10),point(100,10),point(100,200),
        point(50,200), point(50,10)]);

        {On définit le crayon }
        Form1.Canvas.Pen.Color := clBlue;
        Form1.Canvas.Pen.mode := pmNotXor;
        For Count := 0 to (Form1.Width div 5) do
        BEGIN
                {On dessine le triangle }
                Form1.Canvas.PolyLine([point(Count*5,100),point(Count*5+10,100,
                point(Count*5+5,110),point(Count*5,100)]);

                Pause := Time;
                while (Time-Pause) < 1e-12 do; {on ne fait rien}

                {On efface le triangle }
                Form1.Canvas.PolyLine([point(Count*5,100),point(Count*5+10,100,
                point(Count*5+5,110),point(Count*5,100)]);
        end;
end;
```

Le programme commence le dessin d'une boîte rouge en définissant l'état du crayon à pmCopy *et la couleur à* clRed*. Il déplace ensuite un triangle sur l'écran en utilisant le mode de crayon* pmNotXor*. Le mode* pmNotXor *est semblable à XOR car il préserve l'image de fond, mais il affiche la vraie couleur au premier plan. Il est nécessaire de dessiner deux fois le triangle à chacune de ses positions. La première fois, le triangle est dessiné, la seconde fois, il est effacé.*

OBJET PINCEAU ET REMPLISSAGE

Au lieu de n'utiliser que les contours, vous pouvez remplir certains des objets proposés par Delphi. La propriété Brush détermine la façon dont un objet est rempli. Les trois propriétés principales affectant le pinceau (brush) sont Color, Style et Bitmap. On peut utiliser le pinceau de deux façons différentes : avec les propriétés Color et Style ou avec la propriété Bitmap.

Lorsque vous utilisez les propriétés Color et Style, la couleur du remplissage dérive de la valeur de la propriété Color. La propriété Style définit le style du remplissage. De la même façon que vous utilisez la méthode PolyLine pour des objets contourés (non remplis), utilisez la méthode Polygon pour dessiner des polygones remplis. L'exemple de programme suivant montre tous les styles disponibles sur huit triangles différents.

```
procedure Triangle(Iteration : Integer);
begin
        Form1.Canvas.Brush.Color := clBlue;
        Form1.Canvas.Polygon([Point(TRUNC((Iteration/9)*Form1.Width),50),
        Point(TRUNC((Iteration/8)*Form1.Width),100),
        Point(TRUNC(((Iteration-1)/8)*Form1.Width),100),
        Point(TRUNC((Iteration/9)*Form1.Width),50)]);
end;

procedure TForm1.ShowTrianglesClick(Sender: TObject);
begin
        Form1.Canvas.Brush.Style := bsSolid;
        Triangle(1);
        Form1.Canvas.Brush.Style := bsClear;
        Triangle(2);
        Form1.Canvas.Brush.Style := bsHorizontal;
        Triangle(3);
        Form1.Canvas.Brush.Style := bsVertical;
        Triangle(4);
        Form1.Canvas.Brush.Style := bsFDiagonal;
        Triangle(5);
        Form1.Canvas.Brush.Style := bsBDiagonal;
        Triangle(6);
        Form1.Canvas.Brush.Style := bsCross;
        Triangle(7);
        Form1.Canvas.Brush.Style := bsDiagCross;
        Triangle(8);
end;
```

Vous créez une procédure générique qui élabore des triangles dans une des huit régions de l'écran. La procédure ShowTriangleClick *définit divers paramètres pour le pinceau et appelle la procédure de triangle en lui passant la position du triangle souhaité.* Au lieu d'utiliser les styles et les couleurs prédéfinis pour le pinceau, utilisez un bitmap qui définit le motif utilisé par le pinceau pour remplir des objets.

Un bitmap de pinceau est un bitmap de 8 pixels par 8 pixels définissant le motif utilisé pour remplir les objets.

Pour utiliser un bitmap sur un pinceau, vous devez commencer par créer un bitmap, l'affecter, puis le libérer lorsque vous avez terminé. La création et la manipulation des bitmap sont détaillées un peu plus loin.

DESSINER DES RECTANGLES REMPLIS

De même que le type TPoint spécifie un ensemble de coordonnées en Delphi, un type Trect précise une partie rectangulaire dans une zone graphique. Vous spécifiez une région rectangulaire en donnant les coordonnées des coins supérieur gauche et inférieur droit.

La fonction Rect permet de créer un type TRect à partir de coordonnées. La plupart des fonctions manipulant les régions rectangulaires utilisent des types TRect comme paramètres. Ainsi, vous utilisez la méthode FillRect pour dessiner un rectangle rempli. La ligne de code ci-après est un exemple d'utilisation de la méthode FillRect. Remarquez que vous devez utiliser la fonction Rect pour spécifier les coordonnées.

Form1.Canvas.FillRect(Rect(20,20,100,100));

En plus de la procédure FillRect, la procédure Rectangle dessine un rectangle en utilisant les attributs du pinceau courant pour le remplissage et ceux du crayon courant pour les contours. Cependant, cette procédure ne prend pas les mêmes paramètres que la précédente. Les quatre points sont paramètres, et l'on ne passe donc pas un type TRect. La ligne de code ci-après est un exemple d'utilisation de la procédure Rectangle.

Form1.Canvas.Rectangle(20,20,100,100);

CERCLES, COURBES ET ELLIPSES

Tout ce que vous avez pu tracer jusqu'ici était constitué de points distincts ou de combinaisons de lignes droites. Le monde serait bien morne sans courbes ; Delphi propose différentes méthodes pour tracer des cercles, des ellipses, des arcs et des tranches. Un cercle est une ellipse dont le rayon est constant.

Pour dessiner une ellipse en Delphi, il suffit de fournir la région rectangulaire du canevas dans laquelle elle sera contenue. Pour tracer un cercle parfait, il suffit d'exécuter ce qui suit :

Form1.Canvas.Ellipse(100,100,200,200);

Pour dessiner une ellipse dont la largeur est supérieure à la hauteur, exécutez ce qui suit :

Form1.Canvas.Ellipse(100,100,300,200);

Pour ne dessiner qu'une portion d'ellipse, la procédure est un peu plus complexe. La méthode prend huit paramètres. Les quatre premiers sont nécessaires au tracé d'une ellipse complète. Les deux derniers (en fait quatre) sont les points indiquant le pourcentage de l'ellipse qui apparaîtra. Ils représentent les points d'arrivées des deux lignes partant de l'origine et définissant la portion de courbe à tracer. Ainsi, par exemple, la ligne ci-après trace un quart de cercle.

Form1.Canvas.Pie(100,100,200,200,100,100,100,200);

Un arc est en tout point semblable à une tranche, à cela près qu'il n'est pas rempli. Le code ci-après affiche un arc de même longueur que la tranche tracée ci-dessus.

Form1.Canvas.Arc(100,100,200,200,100,100,100,200);

TEXTE

Un objet TFont définit l'apparence du texte. Un objet TFont définit un jeu de caractères à partir d'une hauteur, du nom de famille de la fonte (police), etc. La hauteur est indiquée par la propriété Height, la police par la propriété Name, la taille en points par la propriété Size, la couleur par la propriété Color, et les attributs de la fonte (gras, italique, etc.) par la propriété Style.

Lorsqu'une fonte est modifiée, un événement OnChange se produit.

Outre ces propriétés, ces méthodes et ces événements, cet objet dispose de ceux qui s'appliquent à tous les objets.

les styles

fsBold	La fonte est en gras.
fsItalic	La fonte est en italique.
fsUnderline	La fonte est soulignée.
fsStrikeout	La fonte est affichée avec une ligne horizontale en travers (barrée).

La propriété Style est un ensemble, elle peut donc contenir plusieurs valeurs. Une fonte peut, par exemple, être en gras et en italique.

Pour afficher du texte, appelez la méthode TextOut. Pour déterminer si le texte peut s'afficher à l'intérieur d'une zone définie, utilisez TextHeight et TextWidth.

procedure TextOut(X, Y: Integer; const Text: string);

TextOut dessine la chaîne contenue dans Text sur le canevas en utilisant la fonte courante, l'angle supérieur gauche du texte se situant au point de coordonnées (X, Y).

Exemple
Cet exemple affiche une chaîne texte à la position indiquée sur la fiche lorsque l'utilisateur clique sur le bouton de la fiche:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
        Canvas.TextOut(20, 20, 'DELPHI facilite la programmation Windows');
end;
```

QUELQUES FONCTIONS UTILES :

function TextHeight(const Text: string): Integer;

TextHeight renvoie la hauteur en pixels de la chaîne transmise par Text telle qu'elle apparaîtrait avec la fonte courante. Il est possible d'utiliser TextHeight pour déterminer si la chaîne entière apparaît dans un espace prédéfini.

Exemple
Cet exemple affiche dans une boîte d'édition de la fiche la hauteur d'une chaîne texte telle qu'elle apparaîtrait sur le canevas avec la fonte courante :

```
procedure TForm1.FormCreate(Sender: TObject);
var
        L: LongInt;
begin
        L := Canvas.TextHeight('Object Pascal est le meilleur');
        Edit1.Text := IntToStr(L) + ' est la hauteur en pixels';
end;
```

function TextWidth(const Text: string): Integer;

La méthode TextWidth renvoie la largeur en pixels de la chaîne transmise par le paramètre Text en supposant son affichage avec la fonte courante. Il est possible d'utiliser TextWidth pour déterminer si une chaîne peut être contenue dans un espace préétabli.

Exemple
Cet exemple détermine la largeur de la chaîne indiquée et, si elle est trop large pour s'afficher dans une boîte d'édition, cette dernière est agrandie pour permettre son affichage en entier. Ensuite, la chaîne est affichée dans la boîte d'édition.

```
procedure TForm1.Button1Click(Sender: TObject);
var
        T: Longint;
        S: string;
begin
        S := 'Object Pascal est le langage qu'il me faut';
```

```
        T := Canvas.TextWidth(S);
        if T > Edit1.Width then
        Edit1.Width := T + 10;
        Edit1.Text := S;
end;
```

ONPAINT : POUR REDESSINER UNE FENÊTRE

Autrefois, un programme graphique utilisait tout l'écran et supposait que toute modification à l'écran était le fait des actions du programme lui-même. Dans un environnement Windows 95 ou Windows NT, de nombreuses applications peuvent s'exécuter simultanément sur un écran.

Que se passe-t-il si une fenêtre est recouverte par une autre ou si elle est redimensionnée ?

Le système peut garder une copie de l'écran en mémoire et effectuer en mémoire les modifications qui ne sont pas visibles. Cette méthode serait très coûteuse en ressources, tout particulièrement si de nombreuses applications s'exécutent. Au lieu de cela, le système d'exploitation prévient l'application que quelque chose a changé et c'est à l'application d'y remédier. Dès qu'une mise à jour est nécessaire, un événement OnPaint survient.

Delphi ne redessine que la partie du canevas qui a été affectée ou invalidée. Lorsqu'une partie d'une fiche est invalide, Delphi appelle la procédure spécifiée dans le Gestionnaire d'événements OnPaint*, afin de redessiner la partie invalidée de la fiche.*

L'exemple suivant place dans le Gestionnaire d'événements OnPaint un code qui dessine un quart de cercle dans la fiche. Il fait également apparaître une boîte d'édition affichant le nombre de fois où la fiche a été repeinte.

```
unit unitOnPaint;
interface

uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,Forms, Dialogs,StdCtrls;

type
        TForm1 = class(TForm)
                NumRepaints: TEdit;
                procedure FormPaint(Sender: TObject);
                procedure FormCreate(Sender: TObject);
        private
                { déclarations privées }
                NumPaints : integer;
        public
                { déclarations publiques }
        end;
var
        Form1: TForm1;
```

```
implementation
{$R *.DFM}

procedure TForm1.FormPaint(Sender: TObject);
begin
        NumPaints := NumPaints + 1;
        Form1.Canvas.Pie(100,100,200,200,100,100,100,200);
        NumRepaints.Text := IntToStr(NumPaints);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
        NumPaints := 0;
end;
end.
```

Lorsque vous exécutez ce programme, vous pouvez voir le compteur d'itération augmenter chaque fois que le gestionnaire OnPaint *est appelé. Cela est dû au fait que la variable* NumPaints *est incrémentée et affichée à chaque appel au gestionnaire* OnPaint.

COMPOSANT TPAINTBOX

Tous les dessins que vous avez effectués jusqu'ici l'ont été sur le canevas ou sur une fiche. Il est souvent utile de confiner un graphique à une région rectangulaire d'une fiche. Delphi propose un composant le permettant : TPaintBox.

Exemple :

```
PaintBox1.canvas.Ellipse(0,0,2*PaintBox1.Width,2*PaintBox1.Height);
```

Que se passe t-il ? Vous avez demandé une ellipse mais Delphi n'a tracé qu'un arc. En effet, l'ellipse est plus grande que le composant TPaintBox, qui est la seule zone dans laquelle Delphi peut dessiner. Le reste de l'ellipse a été tronqué. Imaginez la complexité d'une application qui s'assurerait que rien n'est dessiné en dehors d'une région donnée. Le composant TpaintBox s'en charge pour vous. Les coordonnées dans une TPaintBox sont relatives à la TPaintBox elle-même, pas à la fiche. Cela signifie également que tant que le Gestionnaire d'événements OnPaint est responsable du tracé de la TPaintBox, vous pouvez déplacer l'image sur la fiche en modifiant les propriétés Top et Left de la TPaintBox. La TPaintBox utilise la propriété Align afin que l'image reste au sommet, sur la gauche, sur la droite ou au bas de la fiche. Cette propriété oblige également la TPaintBox à remplir la zone client de la fiche.

COMPOSANT TSHAPE : MOINS DE COMPLEXITÉ

Que faire si vous souhaitez manipuler des formes simples, sans pour autant vous préoccuper de gestion d'événements pour contrer l'invalidation ? Existe-t-il des contrôles qui simplifient les actions ? Oui. Le composant TShape encapsule dans ses méthodes et propriétés la plupart

des méthodes de dessin. Le composant TShape a des propriétés représentant son pinceau, son crayon et sa forme. Les formes que peut prendre le composant sont le cercle, l'ellipse, le rectangle, le rectangle arrondi, le carré et le carré arrondi. Le grand avantage de TShape est que tout le code nécessaire au tracé et au réaffichage de l'objet est caché.

PRODUIRE DES IMAGES

Les méthodes et les composants graphiques conviennent parfaitement à la plupart des applications, mais il arrive qu'un développeur souhaite ajouter une image graphique prédessinée à son application. Cela serait assez dur à réaliser si l'on ne disposait que de composants graphiques.

Prenons un exemple : les entreprises Bartlebooth, fabricants de puzzles, veulent placer la photo de leur fondateur sur tous les documents de l'entreprise. Obtenir ce résultat avec les méthodes graphiques serait un vrai cauchemar. Il vous suffit de prendre une photo de John Barnabooth et de la numériser en utilisant un scanner. Le programme de numérisation stocke l'image dans un format particulier que Delphi peut comprendre et se contente d'afficher. Les méthodes ne conviennent pas non plus si un artiste conçoit une image dans un programme de dessin et souhaite l'incorporer dans une application Delphi.

Delphi prend en charge de manière native quatre types d'images : les bitmap, les icônes, les métafichiers et les métafichiers avancés. Ces quatre types de fichiers stockent des images. La différence réside dans la façon dont les images sont stockées dans le fichier et dans les outils permettant de manipuler ces images et d'y accéder. Lorsque vous avez calculé la place prise en mémoire pour stocker une image de l'écran, vous avez dû multiplier la profondeur de couleur (en bits) par la résolution. Un bitmap est une sorte de photographie instantanée de l'écran et de toutes les informations qui lui sont associées. Un bitmap connaît la couleur de chaque pixel de l'image, mais ne sait pas ce que l'image représente. Ainsi, si vous prenez un bitmap d'un carré rouge sur fond bleu, les seuls informations présentes dans le bitmap sont que tous les pixels sont bleus, sauf les pixels qui appartiennent au carré dont les sommets sont (10,10) dans le coin supérieur droit et (100,100) dans le coin inférieur gauche, qui eux sont rouges.

Windows 3.1, Windows NT et Windows 95 ont édicté des formats de fichiers standard pour les bitmap. Les bitmap Windows sont des bitmap indépendants du matériel, ce qui signifie que les informations sont stockées de telle façon que n'importe quel ordinateur peut afficher l'image dans la résolution et avec le nombre de couleurs de sa définition. Cependant, cela ne veut pas dire que l'image a le même aspect sur n'importe quel ordinateur. Le résultat sera bien
meilleur sur un écran acceptant une résolution de 1024 par 768 en couleurs 24 bits que sur un moniteur VGA standard. Le point à retenir est que les utilisateurs de ces deux ordinateurs pourront voir l'image. Ce standard permet au développeur de ne pas se soucier de la signification de chaque octet de ses fichiers bitmap.

Fort heureusement, les détails du format de fichier bitmap sont encapsulés dans le système d'exploitation et dans Delphi. Le moyen le plus facile d'afficher un bitmap en Delphi consiste à utiliser le composant TImage. Ce composant peut afficher différents types d'images

graphiques. Il peut charger un bitmap provenant d'un fichier, et lui servir de conteneur dans l'application. Vous pouvez ainsi distribuer l'application sans devoir inclure un fichier bitmap distinct dans le logiciel.

Les icônes sont en fait de très petits bitmaps. Elles appartiennent à une autre catégorie car elles servent généralement à figurer un raccourci vers une application ou une vue réduite d'objet. De manière interne, les icônes sont stockées comme les bitmap. Les métafichiers et les métafi-chiers avancés, en revanche, sont stockés de façon totalement différente. Un métafichier ne stocke pas des séries de bits décrivant l'image, mais des informations indiquant la façon dont l'image a été créée.

Les métafichiers stockent la séquence de commandes de tracé nécessaire à la re-création de l'image. Pour afficher une image en utilisant le composant TImage, placez un TImage sur une fiche et double-cliquez sur la propriété Picture. Une boîte de dialogue apparaît alors, vous permettant d'afficher, de charger et d'enregistrer un bitmap dans le composant. Cliquez sur Charger et choisissez n'importe quel fichier .pico, .bmp, .emf ou .wmf valide. Le bitmap ou le métafichier que vous avez choisi est alors affiché dans le composant TImage.

ETIRER ET REDIMENSIONNER DES IMAGES

Par défaut, une image est affichée dans sa résolution d'origine, et vous n'en voyez que la partie qui est affichée dans le composant TImage. Deux propriétés importantes affectent la façon dont une image apparaît dans un TImage. La propriété Autosize fait que la taille du composant correspond aux dimensions de l'image. En définissant comme True la propriété Stretch (étirement) du composant, vous obligez l'image à prendre les dimensions du composant. Si vous définissez comme False les propriétés Stretch et Autosize, l'image est par défaut centrée dans le composant. Pour obliger l'image à s'afficher dans le coin supérieur gauche du composant, définissez comme False la propriété Center.

CHARGER UNE IMAGE EN COURS D'EXÉCUTION

Vous avez vu comment utiliser le composant TImage pour afficher un bitmap, un métafichier ou une icône en déclarant l'image au moment de la conception de votre application. Vous pouvez également charger un bitmap provenant d'un fichier en cours d'exécution à l'aide de la méthode LoadFromFile. La ligne qui suit charge le bitmap d'installation de Windows 95 dans votre composant d'image :

Image1.Picture.LoadFromFile('C:\WIN95\SETUP.BMP');

Remarquez que la méthode opère sur la propriété Picture du composant d'image et non sur le composant lui-même. L'image utilise la plupart de ses propriétés pour décrire la façon dont elle interagit avec l'application. La propriété Picture contient des informations sur l'image elle-même, vous devez donc charger l'image dans cette propriété.

CRÉER SON PROPRE BITMAP

Nous vous avons montré précédemment comment dessiner sur le canevas d'une fiche et d'une Paintbox. Est-il possible de dessiner sur le canevas d'un bitmap ? La réponse est oui. En Delphi, un objet TBitmap possède un canevas que vous pouvez manipuler comme le canevas d'une TPaintBox ou d'une TForm. Lorsque vous dessinez sur un bitmap, il n'est pas nécessaire de vous soucier des événements OnPaint pour réafficher la scène en cas d'invalidation. Il vous suffit de recharger le bitmap qui est en mémoire. L'inconvénient du bitmap est qu'il nécessite plus de ressources système car il est stocké en mémoire. En Delphi, un bitmap seul est limité car il est difficile de l'afficher. Cela vient du fait que le bitmap n'est pas lui-même un composant et ne peut donc se "réparer" lui-même si quelque chose s'y inscrit. Cependant, si vous utilisez un bitmap en association avec un composant d'image qui l'affiche, ce composant image répond automatiquement à son propre événement OnPaint en redessinant le bitmap chaque fois que c'est nécessaire.

CRÉER ENTIÈREMENT UN BITMAP

Pour créer un nouveau bitmap, vous devez déclarer une variable de type TBitmap et utiliser la méthode Create comme constructeur pour allouer de l'espace au bitmap.

```
Var
    MyBitmap : TBitmap;

BEGIN
    MyBitmap := TBitmap.Create;
```

Pour l'instant, le bitmap a été créé mais il est encore vide. Il convient maintenant de définir ses dimensions. Utilisez pour cela les propriétés Height et Width :

```
MyBitmap.Height := 100;
MyBitmap.Width := 200;
```

Avant de dessiner le bitmap, ajoutez-lui quelques graphiques (ici, une ligne diagonale).

```
MyBitmap.Canvas.MoveTo(200,100);
MyBitmap.Canvas.LineTo(0,0);
```

Pour afficher un bitmap, vous pouvez utiliser la méthode Draw qui en copie un sur un canevas.

Toute autre manipulation du bitmap s'effectue en mémoire. Pour dessiner le bitmap sur Form1 aux coordonnées 100,100, utilisez la ligne suivante :

```
Form1.Canvas.Draw(100,100,MyBitmap);
```

Lorsque vous avez terminé avec le bitmap, vous devez libérer ses ressources système à l'aide de la méthode Free :

```
MyBitmap.Free;
```

Une méthode pour afficher le bitmap consiste à définir celui que vous avez créé comme l'image d'un composant d'image :

```
Image1.Picture.Graphic := MyBitmap;
```

L'un des avantages de cette méthode est que vous n'avez plus à vous soucier d'une éventuelle invalidation de l'image car le composant se charge de son réaffichage.

DESSINER UN BITMAP

Plusieurs méthode permet de dessiner tout ou partie d'un dessin.

procedure Draw(X, Y: Integer; Graphic: TGraphic);

La méthode Draw dessine sur le canevas le graphique spécifié par le paramètre Graphic au point de coordonnées écran (X, Y) exprimées en pixels. Le graphique peut être un bitmap, une icône ou un métafichier.

Exemple

Lorsque l'utilisateur clique sur Button1, le code suivant dessine le graphique C:\WINDOWS\TARTAN.BMP en le centrant dans Form1. Rattachez ce code au gestionnaire d'événement OnClick de Button1.

```
procedure TForm1.Button1Click(Sender: TObject);
var
        Bitmap1: TBitmap;
begin
        Bitmap1 := TBitmap.Create;
        Bitmap1.LoadFromFile('c:\windows\tartan.bmp');
        Form1.Canvas.Draw((Form1.Width div 2)-(Bitmap1.Width div 2),
        (Form1.Height div 2) - (Bitmap1.Height div 2), Bitmap1);
end;
```

procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);

La méthode StretchDraw dessine le graphique spécifié par le paramètre Graphic dans le rectangle spécifié par le paramètre Rect. Cette méthode permet d'adapter un graphique à la taille du rectangle.

Exemple

Le code suivant adapte le bitmap pour remplir la zone client de Form1 :

```
Form1.Canvas.StretchDraw(Form1.ClientRect, TheGraphic);
```

procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source: TRect; Color: TColor);

La méthode BrushCopy copie une partie de bitmap dans une partie de canevas, en remplaçant l'une des couleurs du bitmap avec le pinceau du canevas de destination. Dest spécifie la partie rectangulaire du canevas de destination où s'effectue la copie. Bitmap indique le graphique source de la copie. Source spécifie la zone rectangulaire du bitmap à copier. Color indique la couleur du Bitmap qui doit être modifiée par le pinceau du canevas (spécifié par la propriété Brush).

BrushCopy peut servir à rendre une image copiée partiellement transparente. Pour ce faire, il suffit de spécifier la couleur de la surface de destination (clBackground par exemple) comme couleur de la propriété Brush du canevas avant d'appeler BrushCopy.

Exemple

Le code suivant montre les différences entre CopyRect et BrushCopy. Le graphique bitmap 'TARTAN.BMP' est chargé dans Bitmap puis affiché sur le canevas de Form1. BrushCopy remplace la couleur noire du graphique au moyen du pinceau du canevas, contrairement à CopyRect qui ne modifie pas ses couleurs.

```
var
        Bitmap: TBitmap;
        MyRect, MyOther: TRect;
begin
        MyRect.Top := 10;
        MyRect.Left := 10;
        MyRect.Bottom := 100;
        MyRect.Right := 100;
        MyOther.Top := 111; {110}
        MyOther.Left := 10;
        MyOther.Bottom := 201; {210}
        MyOther.Right := 100;
        Bitmap := TBitmap.Create;
        Bitmap.LoadFromFile('c:\windows\tartan.bmp');
        Form1.Canvas.BrushCopy(MyRect,Bitmap, MyRect, clBlack);
        Form1.Canvas.CopyRect(MyOther,Bitmap.Canvas,MyRect);
end;
```

procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);

La méthode CopyRect copie dans l'objet canevas une partie d'image d'un autre canevas. La propriété Dest spécifie le rectangle cible du canevas cible. La propriété Canvas spécifie le canevas source. La propriété Source spécifie le rectangle source du canevas source.

Exemple

L'exemple de code suivant copie le bitmap source inversé sur le canevas de Form2 :

```
Form2.Canvas.CopyMode := cmNotSrcCopy;
Form2.Canvas.CopyRect(ClientRect, Canvas, ClientRect);
```

ENREGISTRER UN BITMAP DANS UN FICHIER

Vous pouvez non seulement charger ou manipuler des bitmap, mais aussi les enregistrer dans un fichier. Pour sauvegarder le bitmap, utilisez simplement la méthode SaveToFile.

My bitmap.SaveToFile ('C:\Perso\MyBitmap. BMP') ;

COULEURS ET TRACÉS

LES COULEURS

Les couleurs SYSTEME

clBackground	Couleur courante de fond Windows
clActiveCaption	Couleur courante de la barre de titre de la fenêtre active
clInactiveCaption	Couleur courante de la barre de titre des fenêtres inactives
clMenu	Couleur de fond courante des menus
clWindow	Couleur de fond courante des fenêtres
clWindowFrame	Couleur de fond courante des cadres de fenêtre
clMenuText	Couleur courante du texte des menus
clWindowText	Couleur courante du texte des fenêtres
clCaptionText	Couleur courante du texte dans la barre de titre de la fenêtre active
clActiveBorder	Couleur courante de la bordure de la fenêtre active
clInactiveBorder	Couleur courante de la bordure des fenêtres inactives
clAppWorkSpace	Couleur courante de l'espace de travail de l'application
clHighlight	Couleur de fond courante du texte sélectionné
clHightlightText	Couleur courante du texte sélectionné
clBtnFace	Couleur courante d'une face de bouton
clBtnShadow	Couleur courante de l'ombre projetée par un bouton
clGrayText	Couleur courante du texte grisé
clBtnText	Couleur courante du texte d'un bouton
clInactiveCaptionText	Couleur courante du texte dans la barre de titre d'une fenêtre inactive
clBtnHighlight	Couleur courante d'un bouton en surbrillance

Les couleurs de base

clBlack	Noir	clMaroon	Marron	clGreen	Vert
clOlive	Vert olive	clNavy	Bleu marine	clPurple	Violet
clTeal	Teal	clGray	Gris	clSilver	Argent
clRed	Rouge	clLime	Vert clair	clBlue	Bleu
clFuchsia	Fuchsia	clAqua	Aqua	clWhite	Blanc

LES MODES DE TRACE

La table suivante résume les 16 modes de tracé. Cette table indique comment sont combinées les couleurs du stylo (S) et de la destination (D) pour donner la couleur finale de la destination. La colonne "opération booléenne" utilise la notation 'C' pour exprimer l'opération logique effectuée par Windows.

Le mode par défaut est pmCopy, qui transfère la couleur du stylo dans la destination.
Le mode notCopy trace en noir si le stylo est blanc est en blanc s'il est noir.
Le mode pmNop n'effectue aucune modification de la destination.

Tableau explicitant l'effet de la propriété mode.

Stylo (S)	**1**	**1**	**0**	**0**	**opération booléenne**	**mode de tracé**	**effet**
Destination (D)	**1**	**0**	**1**	**0**			
Résultat	0	0	0	0	0	pmBlack	Toujours noir.
	0	0	0	1	~ (S | D)	pmNotMerge.	Inverse de la combinaison pmMerge de la couleur du crayon et de l'écran
	0	0	1	0	~ S & D	pmMaskNotPen	Combinaison des couleurs communes à l'écran et à l'inverse du crayon.
	0	0	1	1	~ S	pmNotCopy	Inverse de la couleur du crayon.
	0	1	0	0	S & ~ D	pmMaskPenNot	Combinaison des couleurs communes au crayon et à l'inverse de l'écran.
	0	1	0	1	~ D	pmNot	Inverse de la couleur de l'écran.
	0	1	1	0	S ^ D	pmXor	Combinaison des couleurs du crayon et de l'écran mais n'apparaissant pas dans les deux.
	0	1	1	1	~ (S & D)	pmNotMask	Inverse de la combinaison pmMask de la couleur du crayon et de l'écran.
	1	0	0	0	S & D	pmMask	Combinaison des couleurs communes au crayon et à l'écran.
	1	0	0	1	~ (S ^ D)	pmNotXor	Inverse de la combinaison pmXor des couleurs du crayon et de l'écran mais n'apparaissant pas dans les deux.
	1	0	1	0	D	pmNop	Inchangé.
	1	0	1	1	~ S | D	pmMergeNotPen .	Combinaison de la couleur de l'écran et de l'inverse de la couleur du crayon
	1	1	0	0	S	pmCopy	Couleur de crayon spécifiée par la propriété Color.
	1	1	0	1	S | ~ D	pmMergePenNot	Combinaison de la couleur du crayon et de l'inverse de la couleur de l'écran.
	1	1	1	0	S | D	pmMerge	Combinaison de la couleur du crayon et de l'écran.
	1	1	1	1	1	pmWhite	Toujours blanc.

REMPLIR DES FORMES

Windows prévoit deux fonctions permettant de remplir avec une couleur soit un rectangle, soit une zone quelconque pourvue qu'elle soit fermée. La couleur du remplissage est la couleur du pinceau en cours.

procedure FillRect(const Rect: TRect);

La méthode FillRect remplit le rectangle spécifié du canevas en utilisant le pinceau du canevas.

Exemple

Le code suivant crée un rectangle dans le canevas de la fiche et le colorie en rouge en définissant la propriété Brush du canevas à clRed :

```
procedure TForm1.ColorRectangleClick(Sender: TObject);
var
        NewRect: TRect;
begin
        NewRect := Rect(20, 30, 50, 90);
        Form1.Canvas.Brush.Color := clRed;
        Form1.Canvas.FillRect(NewRect);
end;
```

procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);

La méthode FloodFill remplit une zone de la surface de l'écran en utilisant le pinceau en cours (spécifié par la propriété Brush). La méthode FloodFill commence au point de coordonnées (X, Y) et continue dans toutes les directions jusqu'aux limites de couleur.

La manière de remplir la zone est déterminée par le paramètre FillStyle. Si FillStyle est à fsBorder, la surface est remplie jusqu'à ce qu'une bordure de la couleur spécifiée par le paramètre Color soit rencontrée. Si FillStyle est à fsSurface, la surface est remplie tant que la couleur spécifiée par le paramètre Color est rencontrée. Les remplissages fsSurface sont utiles pour remplir une zone ayant une bordure multicolore.

Exemple

Le code suivant remplit de couleur, depuis le centre de la zone client de Form1 jusqu'à ce que la couleur noire soit rencontrée :

```
Form1.Canvas.FloodFill(ClientWidth/2, ClientHeight/2, clBlack, fsBorder);
```

MODES DE TRACÉ GRAPHIQUE

Lorsqu'on dessine dans un contexte graphique, les nouveaux tracés écrasent par défaut les traces existantes. On dit qu'on est dans un mode cmScrCopy, c'est à dire que le tracé sera celui de la couleur du crayon. Mais vous pouvez combiner la couleur du pinceau avec les dessins existants. On parle alors d'opérations logiques entre les pixels du pinceau avec les dessins existants. voici les différents modes de tracé :

property CopyMode: TCopyMode;

La propriété CopyMode détermine le traitement d'une image copiée sur le canevas à partir d'un autre canevas. La valeur par défaut cmSrcCopy de CopyMode signifie que les pixels du canevas source sont copiés sur le canevas cible en remplaçant l'image qui s'y trouve. En changeant CopyMode, il est possible de créer des effets spéciaux. Le tableau suivant donne la liste des valeurs possibles de CopyMode, avec leur description:

cmBlackness	Noircit l'image en sortie.
cmDstInvert	Inverse le bitmap de destination.
cmMergeCopy	Combine le motif et le bitmap source en utilisant l'opérateur booléen AND.
cmMergePaint	Combine et inverse le bitmap source avec le bitmap de destination en utilisant l'opérateur booléen OR.
cmNotSrcCopy	Copie le bitmap source inversé vers le bitmap de destination.
cmNotSrcErase	Inverse le résultat de la combinaison des bitmaps source et de destination en utilisant l'opérateur booléen OR.
cmPatCopy	Copie le motif dans le bitmap de destination en utilisant l'opérateur booléen XOR.
cmPatInvert	Combine le bitmap de destination avec le motif en utilisant l'opérateur booléen XOR.
cmPatPaint	Combine le bitmap source inversé avec le motif en utilisant l'opérateur booléen OR. Combine ensuite le résultat de cette opération avec le bitmap de destination en utilisant l'opérateur booléen OR.
cmSrcAnd	Combine les pixels des bitmaps source et de destination en utilisant l'op. booléen AND.
cmSrcCopy	Copie le bitmap source dans le bitmap de destination.
cmSrcErase	Inverse le bitmap de destination et combine le résultat avec le bitmap source en utilisant l'opérateur booléen AND.
cmSrcInvert	Combine les pixels des bitmaps source et de destination en utilisant l'op. booléen XOR.
cmSrcPaint	Combine les pixels des bitmaps source et de destination en utilisant l'opérateur booléen OR.
cmWhiteness	Blanchit le bitmap en sortie.

Exemple

```
Form2.Canvas.CopyMode := cmNotSrcCopy;
Form2.Canvas.CopyRect(ClientRect, Canvas, ClientRect);
```

CODAGE DE LA COULEUR :

Sous Windows, toute couleur est obtenue en mélangeant les trois couleurs de base, dites couleurs primaires.

Le système RGB (Red-Green-Blue) utilisé en vidéo définit une couleur quelconque comme un ensemble de valeurs de 0 à 255 correspondant à une intensité respective des 3 couleurs primaires. Il permet de distinguer 255^3 couleurs, soit environ 16 millions de couleurs.

Dans Windows, une valeur RGB est représentée sous la forme d'un entier long (4 octets). Les intensités du rouge, du vert et du bleu sont stockées chacune sur un octet, le quatrième octet n'étant pas utilisé. DELPHI utilise lui un type particulier, le type TColor, pour coder les couleurs.

Il faut donc utiliser des fonctions permettant de convertir les valeurs RGB en valeurs TColor et réciproquement.

ColorToRGB convertit un type TColor en valeur RGB
RGB calcule une valeur RGB à partir des intensités des couleurs primaires

GetBValue retourne l'intensité du bleu à partir d'une valeur RGB
GetGValue retourne l'intensité du vert à partir d'une valeur RGB
GetRValue retourne l'intensité du rouge à partir d'une valeur RGB

Il n'y a pas de fonction permettant de convertir directement une valeur RGB en TColor.

INCORPORATION D'IMAGES

Il est possible d'incorporer des images bitmap dans certains objets. En particulier dans les objets de type "boite de liste" en lieu et place des chaînes de caractères habituelles.

Liste d'images :

Pour qu'une boite de liste contienne des images il faut tout :

1. Initialiser la propriété style à la valeur lbOwnerDrawFixed ou lbOwnerDrawVariable. Dans le premier cas, la hauteur de chaque item est identique, déterminée par la propriété ItemHeight. Dans le deuxième cas, elle doit être indiquée pour chaque item dans l'événement OnMeasureItem.

2. Créer un gestionnaire d'événement avec l'événement OnDrawItem. On peut alors dessiner ce que l'on veut sur le canevas. Il faut néanmoins respecter le cadre du rectangle affecté à l'item. Ce rectangle est passé en argument à la procédure événementielle.

Les données nécessaires au dessin sont généralement stockées directement dans les items.

Rappelons que la propriété items est un objet de type Tstrings : il est alors possible d'associer à chaque chaîne un objet, par exemple de type bitmap. Pour un indice i donné, la chaîne est items.strings[i] et l'objet items.objects[i].

Uniquement à l'exécution.

La propriété Objects permet, uniquement à l'exécution, d'accéder à un objet de la liste d'objets associée à la liste de chaînes. Chaque chaîne d'une liste de chaînes a un objet associé.

Le plus souvent, les objets d'une liste de chaînes et d'objets permettent d'associer des bitmaps aux chaînes afin d'utiliser les bitmaps dans des contrôles dessinés par le propriétaire.

Par exemple, dans une boîte liste dessinée par le propriétaire, il est possible d'ajouter la chaîne 'Banane' et une image bitmap de la banane à la propriété Items de la boîte liste en utilisant la méthode AddObject. Il est alors possible d'accéder à la chaîne 'Banane' en utilisant la propriété Strings et au bitmap en utilisant la propriété Objects.

La valeur du paramètre Index permet de spécifier la position dans la liste (l'indice) de l'objet auquel accéder. L'indice étant à base zéro, l'indice du premier objet de la liste est 0, l'indice du deuxième objet est 1, etc.

Pour associer un objet à une chaîne existante, il faut affecter l'objet à la propriété Objects en utilisant le même indice que celui de la chaîne existante dans la propriété Strings. Si, par exemple, un objet chaîne nommé Fruits contient la chaîne 'Banane' et s'il existe une image bitmap d'une banane nommée BananaBitmap, il est possible de faire l'affectation suivante :

Fruits.Objects[Fruits.IndexOf('Banane')] := BananaBitmap;

Exemple

Le code suivant permet à l'utilisateur de spécifier un fichier bitmap avec le composant boîte de dialogue d'ouverture OpenDialog1 lors de la création de Form1. Le fichier bitmap spécifié est ajouté à la liste Items de ListBox1. Si ListBox1 est un contrôle dessiné par le propriétaire (un contrôle dont la propriété Style est lbOwnerDrawFixed ou lbOwnerDrawVariable), la deuxième procédure est le gestionnaire d'événement OnDrawItem de ListBox1. Le bitmap de la propriété Object et le texte d'un élément sont obtenus et affichés dans Listbox1

```
procedure TForm1.FormCreate(Sender: TObject);
var
        TheBitmap: TBitmap;
begin
        if OpenDialog1.Execute then
        begin
                TheBitmap := TBitmap.Create;
                TheBitmap.LoadFromFile(OpenDialog1.FileName);
                ListBox1.Items.AddObject(OpenDialog1.FileName, TheBitmap);
        end;
end;

procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index:
Integer; Rect: TRect; State: TOwnerDrawState);
var
        DrawBitmap: TBitmap;
begin
        DrawBitmap := TBitmap(ListBox1.Items.Objects[Index]);
        with ListBox1.Canvas do
        begin
                Draw(Rect.Left, Rect.Top + 4, DrawBitmap);
                TextOut(Rect.Left + 2 + DrawBitmap.Width, Rect.Top + 2,
                        ListBox1.Items[Index]);
        end;
end;
```

LES DLL

Une DLL (Dynamic-Link Library = Librairie dynamique) est un module exécutable constitué de fonctions et de ressources. Ce module forme un fichier, qui a généralement l'extension DLL, que l'on peut charger en mémoire à partir d'une application. Une fois chargé, les routines qu'il contient sont exécutée par l'application comme si elles faisaient partie de son code.

L'utilisation des DLL apporte les avantages suivants :

- La DLL n'est chargée que lorsque l'on a besoin des fonctions qu'elle contient. Mais cet avantage n'en est un que dans le cas où l'on n'utiliserait pas les possibilités de chargement dynamique des fenêtres de DELPHI et que les DLL constituées ne soient pas trop importantes.

- Comme le concept des Unités, dans le monde du Pascal, on peut partager le code d'une DLL entre plusieurs applications. Il s'agit là de l'avantage prépondérant : si une DLL est constituée de fonctions utilisées par plusieurs applications il n'y a qu'un chargement en mémoire. D'où une économie certaine. Cet avantage n'en est un que si l'on ne crée pas des DLL spécifiques à chaque application.

- Le format d'une DLL est standardisé : on peut donc appeler des DLL créées dans un autre langage. On peut de même placer dans une DLL des fonctions qui ne sont pas fournies par un environnement de développement donné. Lors de la programmation d'une application l'appel de la DLL suppléera à ces manques.

  Il faut bien entendu que les paramètres demandés par les fonctions de la DLL et, le cas échéant, les résultats renvoyés aient des types compatibles avec ceux des autres langages.

Par exemple, si l'on souhaite utiliser une DLL écrite avec DELPHI dans un environnement utilisant le langage C, on peut utiliser des variables de types Char, Integer, etc... mais pas Boolean.

Il existe différents types de DLL (mais elles sont souvent un mélange des trois):

- DLL de fonctions : c'est le cas général de bibliothèques partagées.
- DLL de données partagées : plusieurs applications peuvent se partager ces données.
- DLL de ressources : ces DLL ne contiennent que des ressources graphiques (bitmaps, icônes, curseurs, etc...).

FONCTIONNEMENT D'UNE DLL

Lorsqu'une DLL est appelée pour la première fois par une application, elle est chargée en mémoire.

Elle gère alors un compteur qui s'incrémente à chaque fois qu'elle est recherchée par une application. Ce compteur se décrémente lorsqu'une application cesse de l'utiliser.

Lorsque le compteur revient à zéro, la DLL est déchargée de la mémoire.

Mode d'appel d'une DLL :

Le mode d'appel des fonctions d'une DLL est celui standardisé par Windows, c'est à dire celui du Pascal (les paramètres sont placés dans la pile mais c'est l'application appelante qui vide la pile au retour de la fonction) et non celui du langage C (c'est la fonction appelée qui vide la pile).

Ce qui fait qu'il n'est pas nécessaire de modifier les prototypes des fonctions d'une DLL (alors que les DLL construites en C doivent utiliser le mot clé 'Pascal' pour qu'elles soient appelées selon le mode Pascal).

Pour créer une DLL il faut reprendre en grande partie la procédure de création d'un projet. On peut donc créer une DLL à partir de UNIT1.PAS et PROJETC1.DPR proposé au démarrage par DELPHI.

Bien évidemment un "projet" DLL peut contenir plusieurs unités. Il peut n'être constitué que de fichiers '.PAS' (dans ce cas il vaut mieux ouvrir le menu 'Fichier | Nouvelle unité ') ou contenir des interfaces (dans ce cas, il faut appeler 'Fichier | Nouveau Projet ').

Modification du fichier PROJECT1.DPR :

L'essentiel des modifications à apporter au projet pour qu'il soit compiler en tant que DLL se situe dans le fichier source du projet (d'extension .DPR)

- Changer le mot réservé **'Program'** par **'Library'**.
- Supprimer le mot **'Forms'** et les lignes qui en dépendent de la clause **'Uses'**.
- Ajouter la clause **'Exports'** après la clause **'Uses'**.
- Supprimer tout ce qui se trouve dans le bloc d'initialisation (**Application.Run, etc**).

La section **'Exports'** contiendra le nom de toutes les routines exportées, c'est à dire celles qui peuvent être appelées à partir d'une application.

> Au sein d'une DLL il peut y avoir des fonctions et procédures internes qui ne seront pas exportées.

La DLL prendra le nom de la librairie. Comme d'habitude il faut laisser à DELPHI le soin de modifier lui-même ce nom en sauvegardant le projet.

Voilà le squelette d'une DLL :

```
library Madll;
uses
        Dll1 in 'DLL1.PAS'   {FDLL};

{$R *.RES}
begin
end.
{ La clause Exports sera rajoutée ultérieurement }
```

Modification dans les unités :

Dans les unités contenant les diverses fonctions de la DLL les modifications sont réduites à l'ajout du mot réservé à la fin de l'en-tête décrivant les fonctions exportées dans la section 'interface' :

Dans l'exemple précédent on a alors :

```
type
        TFDLL = class(TForm)
        private
        { Private-déclarations }
        public
        { Public-déclarations }
        end;
        Function Puissance ( valeur , exposant : longint ) : longint ; Export ;stdcall ;
```

La déclaration de la fonction, dans la section 'implementation', reste inchangée :

```
Function Puissance (valeur, exposant : longint ) : longint ;stdcall ;
var
        i : longint ;
        transit : longint ;
begin
         case exposant of
                0 : result := 1 ;
                1 : result := valeur ;
                else
                begin
                i := 2 ;
                        transit := valeur ;
                        while ( i <= exposant ) do
                        begin
                                transit := transit * valeur ;
                                inc ( i ) ;
                        end ;
                        result := transit ;
                end ;
        end;
end ;
```

Il faut alors revenir dans le source du projet (.DPR) pour rajouter la clause 'Exports', sous la clause 'Uses', et y indiquer le nom de la fonction exportée.

```
exports
    Puissance ;
```

Il faut enfin compiler le projet pour créer la DLL. Pour cela il faut passer par le menu **'Compiler | Tout construire** ' (le menu 'Exécuter' ne servant à rien dans ce cas).

La DLL, au nom du projet et à l'extension .DLL est créée.

MISE EN PLACE ET CHARGEMENT D'UNE DLL :

Mise en place :
Normalement une DLL est placée dans le répertoire où se trouve l'exécutable de l'application qui l'utilise. Elle est alors directement accessible.

Si l'on souhaite que la DLL soit partagée par plusieurs applications il faut positionner celle-ci dans un répertoire où Windows la recherchera.

Lorsqu'un programme appelle une DLL, Windows effectue la recherche dans l'ordre suivant:
- Répertoire de l'application appelante ;
- Répertoire Windows ;
- Sous-répertoire System32 de Windows ;
- Chemins du PATH.

Il est donc préférable de positionner une DLL partageable dans le répertoire Windows ou son sous-répertoire System32.

Chargement d'une DLL :
Il est possible de charger une DLL selon deux modes :

Le mode statique :
Dans ce mode, le nom de la DLL à charger est indiqué explicitement dans le source de l'application appelante. Le chargement a lieu au lancement de l'application.

Le mode dynamique :
Dans ce mode, plus complexe à mettre en œuvre, les DLL ne sont chargées qu'au cours de l'exécution, si elles sont nécessaires.

Un des avantages attendus de l'utilisation des DLL est annulé lorsqu'on réalise un chargement statique.

Néanmoins une DLL chargée statiquement reste partageable.

CHARGEMENT STATIQUE D'UNE DLL

Une fonction définie dans une DLL doit être déclarée par l'application appelante avant d'être utilisée.

Cette déclaration doit être réalisée dans la partie 'implémentation' d'une unité avant la définition des fonctions qui la constitue. La syntaxe de cette déclaration est la suivante :

< Prototype de la fonction appelée > ; stdcall ; external < nom de la DLL >

- La déclaration du prototype est faite selon les règles habituelles ;
- Le mot réservé 'stdcall' défini la convention d'appel, c'est à dire le protocole utilisé pour le passage des variables à une fonction. Pour assurer la compatibilité avec Win32, il est conseillé d'utiliser ***'stdcall'*** ;
- Le nom de la DLL doit être écrit avec son extension.

Si l'on crée un projet contenant une feuille permettant de calculer la valeur de la puissance d'un nombre (3 zones d'édition : une pour entrer la valeur, l'autre l'exposant, la troisième affichant le résultat après le calcul). On a la déclaration suivante :

```
implementation
{$R *.DFM}

Function Puissance( valeur , exposant : longint ): longint ; far ;
        external 'MADLL.DLL';
.....
```

L'appel de la fonction appartenant à la DLL se fait alors de manière classique au sein d'un gestionnaire d'événement :

```
procedure TForm1.Button1Click ( Sender : TObject) ;
var
        Calcul : longint ;
begin
        Calcul:=Puissance(StrToInt ( EValeur.Text ) , StrToInt ( EExposant.Text ));
        EResultat.Text := IntToStr ( Calcul ) ;
        EValeur.Clear ;
        EExposant.Clear ;
        EValeur.SetFocus ;
end ;
```

La méthode précédente correspond à une importation 'par le nom'. Il est néanmoins possible d'importer une fonction par son numéro d'index (c'est à dire le nombre indiquant son positionnement dans la liste des fonctions constituant la DLL).

On utilise alors la syntaxe de déclaration :

< Prototype de la fonction > ; far ; external 'NOM_DLL' index numéro ;

Exemple :
Function Puissance(valeur,exposant:longint):longint;far;external 'MADLL.DLL' index 1;

CHARGEMENT DYNAMIQUE D'UNE DLL

Pour réaliser un chargement dynamique d'une DLL, plus performant, nécessite l'utilisation de deux fonctions de l'API Windows : **LoadLibrary ( )** et **GetProcAddress ( ).**

Function LoadLibrary (LibFileName: PChar): THandle;

Cette fonction charge une DLL, spécifiée par un pointeur sur une chaîne de caractères AZT, en mémoire afin que les fonctions qu'elle contient puissent être invoquées.

Si le chemin spécifié n'est pas complet Windows recherche la DLL dans les répertoires indiqués précédemment.

Le compteur d'utilisation de la DLL est incrémenté. Si la DLL est déjà chargée en mémoire, il n'y a pas de nouveau chargement mais simplement incrémentation du compteur.

Function GetProcAddress (Module: THandle; ProcName: PChar):TFarProc;

Cette fonction renvoie l'adresse de la DLL spécifiée par Module.

ProcName est une chaîne AZT contenant le nom de la fonction appelée (ou alors indique son numéro d'index).

Renvoie NIL si échec.

Procedure FreeLibrary (LibModule : THandle) ;

Cette fonction décrémente le compteur de chargement de la DLL spécifiée par le handle LibModule.

Quand le compteur est à zéro, la DLL est déchargée de la mémoire (la zone mémoire occupée est rendue libre.

La procédure à respecter, pour appeler dynamiquement une DLL est la suivante :

1- Déclarer un type particulier correspondant au prototype de la fonction à appeler puis une variable à ce type. (en fait, il s'agit de déclarer un pointeur sur une fonction d'un type donné).

Dans l'exemple précédent on déclare ainsi le type suivant :

```
type
        TFonctionDLL= Function ( val , exp : longint ) :longint ;
var
        UneFonctionDLL : TFonctionDLL ;
```

2- Charger la DLL dans le gestionnaire d'événement approprié et y rechercher la fonction souhaitée :

```
procedure TForm1.Button1Click ( Sender : TObject) ;
type
        TFonctionDLL= Function ( val , exp : longint ) : longint ;
var
        UneFonctionDLL : TFonctionDLL ;
        Calcul : longint ;
        HandleDLL : THandle ;
begin
        HandleDLL := LoadLibrary ( 'MaDLL.dll ' ) ;
        @UneFonctionDLL := GetProcAddress ( HandleDLL, 'Puissance' ) ;
        if @UneFonctionDLL <> Nil then
         Calcul := UneFonctionDLL ( StrToInt ( EValeur.Text ),
                        StrToInt ( EExposant.Text )) ;
        FreeLibrary ( HandleDLL ) ;
        EResultat.Text := IntToStr ( Calcul ) ;
        EValeur.Clear ;
        EExposant.Clear ;
        EValeur.SetFocus ;
end ;
```

{ L'opérateur '@' utilisé dans l'expression '@UneFonctionDLL' permet de ne pas typer la variable (c'est à dire se conformer à ce qui est renvoyé par GetProcAddress ()}

DLL GRAPHIQUE

CRÉATION D'UNE NOUVELLE DLL

Exemple :création d'une DLL graphique de saisie de date valide

Fichier date.dpr

```
library Date;

uses
   {utilise l'unité unit1}
   Unit1 in 'UNIT1.PAS' {dateinfo};

exports
     choix_date;

{$R *.RES}

begin
end.
```

Seul choix_date est utilisable à partir d'une application

Fichier unit1.pas

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, Grids, Calendar;

type
  Tdateinfo = class(TForm)
    mois: TComboBox;
    annee: TComboBox;
    calendrier: TCalendar;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;

    procedure FormCreate(Sender: TObject);     {création de la fiche}
    procedure moisChange(Sender: TObject);     {changement de mois}
    procedure anneeChange(Sender: TObject);    {changement d'année}
  private
    {méthode appelée lors de l'exécution de la DLL}
    function getdate :string;
  public

  end;
```

```
const
    {pour plus de détail voir function FormatDateTime(): string;}
    slongmonthnames = 65488;
    sshortmonthnames = 65472;
    sshortdaynames = 65504;
    slongdaynames = 65511;

var
  dateinfo: Tdateinfo;

  {la procédure getdate est exportable}
   {=============================================================
}
  procedure choix_date(handle : thandle; var s: string);export;
   {=============================================================
}

implementation
{$R *.DFM}

{création de la fenetre graphique}
procedure Tdateinfo.FormCreate(Sender: TObject);
var
   i : integer;
begin
    {chargement des combobox}
    for i:=0 to 11 do  mois.items.add(loadstr(slongmonthnames+i));

    for i:=1990 to 2010 do annee.items.add(inttostr(i));

    mois.itemindex:=calendrier.month -1;
    annee.itemindex:=annee.items.indexof(inttostr(calendrier.year));
end;

{changement du mois}
procedure Tdateinfo.moisChange(Sender: TObject);
begin
    calendrier.month:=mois.itemindex +1;
end;

{changement de l'année}
procedure Tdateinfo.anneeChange(Sender: TObject);
begin
    calendrier.year:=strtoint(annee.text);
end;

{getdate est une méthode de dateinfo qui renvoie la date choisie}
function Tdateinfo.getdate : string;
begin
    result := inttostr(calendrier.day)+'/'+inttostr(calendrier.month)+'/'+
```

```
        inttostr(calendrier.year);
end;
{ primitive de la DLL}
{utilise la handle de l'application et met la date dans un chaine de caractères}
procedure choix_date(handle : thandle; var s: string);
begin
    {récupération du handle}
    {La propriété Handle donne accès au descript. de fenêtre de l'application}
    application.handle:=handle;

    {création de la forme}
    dateinfo:=tdateinfo.create(application);

    {récupération du choix}
    if dateinfo.showmodal<>mrcancel then
      s:=dateinfo.getdate
    else
       s:='inconnu';

    {libération des ressources}
    dateinfo.free;
end;
end.
```

APPEL DE LA DLL

Cette application fait appel à la DLL graphique pour saisir une date valide

```
{utilisation d'une DLL graphique}
unit Unite;

interface

uses
        SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
        Forms, Dialogs, StdCtrls, Buttons;

type
        TForm1 = class(TForm)
                BitBtn1: TBitBtn;
                Edit1: TEdit;
                procedure BitBtn1Click(Sender: TObject);
        private
        { Déclarations private }
        public
        { Déclarations public }
        end;

var
        Form1: TForm1;
```

```
implementation
{$R *.DFM}

{chargement statique d'une DLL}
procedure choix_date( handle : thandle; var s : string);stdcall;external 'date';

{appel de la primitive de la dll}
procedure TForm1.BitBtn1Click(Sender: TObject);
var
        chaine :string;
begin
        choix_date(application.handle,chaine);
        edit1.text:=chaine;
end;

end.
```

Chargement statique de la DLL

On passe le handle de l'application en paramètre et on récupère la réponse dans chaine.

CRÉATION DE COMPOSANTS

A mesure que le logiciel devient plus sophistiqué et complexe, l'utilisation de composants s'impose pour le développement d'applications. Delphi est un outil de développement parfait pour créer des composants servant à n'importe quelle application logicielle qui en utilisent.

Vous pouvez ainsi développer des composants visuels Delphi natifs et ActiveX. Les premiers peuvent être construits et installés dans l'EDI Delphi, sans les complications associées aux ActiveX. Les seconds peuvent être importés dans Delphi, Visual Basic, dans des pages Web, et dans tout autre produit capable de les incorporer.

Nous commencerons par un aperçu des avantages liés à l'utilisation de composants, puis nous verrons comment créer des composants visuels en Delphi, en commençant par un exemple très simple pour finir sur des composants utiles et puissants. Nous verrons ensuite la manière dont Delphi peut convertir des composants visuels en ActiveX.

POURQUOI ÉCRIRE DES COMPOSANTS ?

La puissance des composants tient au fait qu'ils cachent au développeur tous les détails de l'implémentation. De même qu'un conducteur n'a pas besoin, pour conduire, de connaître les principes thermodynamiques à l'œuvre dans un moteur à explosion, un développeur d'application n'a pas besoin de savoir comment fonctionne un composant pour l'utiliser. Il lui suffit de savoir comment communiquer avec lui. Il y a au moins quatre bonnes raisons d'écrire vos propres composants.

RÉUTILISER LE CODE

L'interface vers un composant est intégrée à l'environnement de développement Delphi. Par conséquent, si vous utilisez fréquemment un objet, vous pouvez le transformer en composant visuel, l'ajouter à la barre d'outils, ce qui en facilite l'utilisation et en cache l'implémentation au sein de la bibliothèque.

MODIFIER LES COMPOSANTS VISUELS EXISTANTS

Comme Delphi est un langage orienté objet et comme un composant visuel est un objet, vous pouvez créer un composant visuel comme sous-classe d'un composant déjà existant. Supposons, par exemple, que vous utilisiez souvent des cercles bleus dans vos applications. Vous voulez tirer parti de toutes les propriétés, événements et méthodes du composant TShape, mais vous prenez toujours un cercle comme paramètre pour la forme et la couleur bleue. Vous pouvez dès lors créer une sous-classe de TShape appelée TCercleBleu. La classe TcercleBleu aura, par défaut, la propriété shape égale à circle et la propriété color égale à Blue.

VENDRE DES COMPOSANTS

Si vous devez ajouter des fonctionnalités spécialisées à votre application, vous pouvez acheter un composant à un éditeur indépendant. C'est l'un des plus grands avantages d'un système basé sur les composants. Ainsi, si vous avez besoin de fonctions de réseau, vous pouvez écrire vous-même les routines afférentes ou acheter un ensemble de composants qui se chargent de ces fonctionnalités. De même, vous pouvez créer un composant et le vendre à d'autres développeurs.

Lorsque vous vendez la bibliothèque, vous ne fournissez qu'une version compilée du produit. Votre client ne voit donc pas son code source ou les détails de son implémentation. Cependant, lorsque le composant est ajouté à une fiche, votre acheteur peut communiquer avec le composant par le biais de l'IDE.

MODIFICATIONS DE COMPORTEMENT

Si vous compilez du code pour en faire un composant visuel, vous pouvez voir son aspect se modifier au cours du développement. Ainsi, si vous définissez comme un cercle la propriété Shape du composant TShape, la forme sur la fiche se transforme en cercle, et ce, avant que le composant ne soit compilé. Cela est très utile dans le cas de composants qui sont visibles dans l'application. Vous pouvez ainsi savoir à quoi celle-ci ressemblera une fois compilée.

CONSTRUIRE ET INSTALLER UN COMPOSANT

Il est facile de créer des DLL, mais elles ne s'intègrent pas à l'EDI de Delphi. Elles sont utiles, mais ne sont pas orientées objets. Les composants visuels ne comportent pas ces deux défauts.

Le premier que vous allez construire dans cette partie se compile simplement et s'installe ou s'enlève de la barre d'outils. Vous pouvez aussi l'ajouter à une fiche, mais il ne fera strictement rien.

Dans Delphi, les composants sont compilés sous forme de paquets. En conséquence, la première chose à faire est de créer un nouveau paquet. Choisissez Fichier, Nouveau pour faire apparaître la boîte de dialogue Nouveaux éléments, puis sélectionnez l'icône "paquet". Un nom de fichier vous sera alors demandé. Dans les exemples qui suivent, nous utiliserons le nom tdcomps. Vous devriez maintenant voir une boîte de dialogue Gestionnaire de paquet vierge.

AJOUTER LE COMPOSANT TDONOTHING AU PAQUET

Dans cet exemple, nous allons utiliser l'Expert composant pour générer le code nécessaire à la création du squelette d'un composant. Tous les composants doivent être issus d'autres composants.

Si vous souhaitez partir de zéro, vous devez créer une sous-classe de TComponent, comme suit :

1. Cliquez sur l'icône Ajouter dans la boîte de dialogue Gestionnaire de paquet. La boîte.Ajouter apparaît alors.
2. Sélectionnez l'onglet Nouveau composant.
3. Spécifiez TComponent comme type d'Ancêtre, TDoNothing comme Nom de classe et Exemples comme Page de palette. Pour le nom du fichier d'unité, choisissez un nouveau fichier .pas qui sera utilisé pour le code source, et laissez la valeur par défaut dans le Chemin de recherche.
4. Cliquez sur OK pour enregistrer ces paramètres.

Delphi crée le noyau d'une unité qui se compile en un composant visuel. En double-cliquant sur celle-ci dans le Gestionnaire de paquet, vous faites apparaître le code source.

Squelette d'un composant visuel

```
unit DoNothing;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
    TDoNothing = class(TComponent)
    private
        { Déclarations privées }
    protected
        { Déclarations protégées }
    public
        { Déclarations publiques }
    published
        { Déclarations publiées}
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Samples', [TDoNothing]);
end;
end.
```

En temps normal, vous modifiez le noyau pour ajouter vos propres fonctionnalités au composant. La bibliothèque de composants est constituée de la classe TDoNothing *et de la procédure* Register. *Delphi utilise tout le code associé à la classe pour déterminer le fonctionnement du composant. Il appelle également la procédure* Register *pour placer le composant sur la barre d'outils. En l'occurrence, Delphi se contente de l'installer sur l'onglet Exemples.*

COMPILER ET INSTALLER PAQUET ET COMPOSANTS

Pour compiler et installer le composant, suivez ces étapes :

1. Dans le Gestionnaire de paquet, cliquez sur l'icône Compiler. L'unité sera alors compilée dans le fichier de paquet DPK.
2. Dans le Gestionnaire de paquet, cliquez sur l'icône Installer. Le paquet sera alors installé. Si son installation fonctionne bien, vous en serez averti par une boîte de dialogue.

Regardez la barre d'outils Exemples. Vous pouvez voir qu'un nouveau composant y est apparu. Placez le pointeur de la souris dessus pour qu'une info-bulle apparaisse. Le nouveau composant est DoNothing. Créez une nouvelle application et ajoutez-le à la fiche. Lorsque vous cliquez dessus pour visualiser les pages de propriétés et d'événements, vous voyez qu'il n'y a que deux propriétés. La propriété Name est DoNothing1 par défaut et Tag est 0 par défaut. La page d'événements n'en contient aucun. C'est un composant visuel, dont la fonction est de ne rien faire.

.

ENLEVER LE COMPOSANT

Il arrive que vous ayez besoin de retirer un composant de la barre d'outils ou de le désinstaller de Delphi. Cela peut être le cas si vous recevez une version mise à jour d'un composant visuel, ou si vous souhaitez simplement enlever un composant dont vous n'avez pas l'utilité. TdoNothing est par essence inutile. Pour l'enlever, suivez ces étapes :

1. Sélectionnez Projet, Options dans le menu, puis sélectionnez l'onglet Paquets.
2. Dans cet onglet, cliquez sur le paquet que vous venez d'installer, puis cliquez sur Supprimer. Le composant TDoNothing est alors retiré de la barre d'outils.

ECRIRE UN COMPOSANT VISUEL

Un développeur d'applications utilise des composants en les plaçant sur une fiche ou en employant la méthode Create. Une fois le composant créé, vous pouvez le manipuler en définissant des propriétés, en appelant des méthodes et en répondant à des événements. Vous n'avez pas à vous soucier de la façon dont fonctionnent les propriétés et les méthodes. Le composant appelle son Gestionnaire d'événements lorsqu'un événement particulier survient. Lorsque vous écrivez un composant, vous devez détailler l'implémentation pour les propriétés et les méthodes, et appeler des Gestionnaires d'événements. Pour ce faire, définissez une classe qui devient le composant et complétez les différents éléments.

LES DIFFERENTS TYPES DE DÉCLARATIONS

Delphi utilise des classes d'objet pour créer des composants visuels. Les différentes parties de la définition de la classe d'objet sont déclarées dans plusieurs régions protégées. Les variables, les procédures et les fonctions peuvent avoir quatre types d'accès :

Privé (Private)	Seules les procédures et les fonctions définies dans la définition de classe ont un accès, et seules les routines se trouvant dans la même unité en ont également un.
Protégé (Protected)	Les procédures et fonctions définies dans la définition de classe, ainsi que les procédures et fonctions des classes descendantes, en ont un.
Public (Public)	Toutes les procédures et fonctions ont un accès.
Publié (Published)	Accès public avec un branchement sur l'EDI de Delphi permettant d'afficher les informations dans les pages propriétés et événements.

Propriétés

Un développeur Delphi utilise les propriétés d'un composant pour lire ou modifier certains attributs, qui sont similaires aux champs de données stockés dans une classe. Les propriétés peuvent cependant provoquer l'exécution de code. Ainsi, lorsque vous modifiez la propriété Shape du composant TShape, ce dernier change de forme. Il existe un mécanisme qui lui ordonne de se modifier lorsque la propriété change. Autrement dit, une propriété peut endosser deux rôles. Ce peut être une donnée affectant le fonctionnement d'un composant, ou le déclencheur d'une action.

Méthodes

Les méthodes sont des procédures et des fonctions qu'une classe a rendues publiques. Bien que vous puissiez utiliser des propriétés pour appeler une fonction ou une procédure, ne le faites que si c'est logique. En revanche, les méthodes peuvent être utilisées n'importe quand. Celles-ci peuvent accepter plusieurs paramètres et renvoyer des données par le biais de déclarations de variables VAR, alors que les propriétés sont définies avec une donnée.

Evénements

Les événements permettent au développeur ou à l'utilisateur d'améliorer le composant lorsqu'un événement survient. Ainsi, l'événement OnClick signifie "Si vous voulez faire quelque chose lorsque l'utilisateur clique ici, dites-moi quelle procédure exécuter pour cela". C'est le travail du concepteur d'appeler les événements du composant si nécessaire.

CONSTRUIRE UN COMPOSANT

Nous allons maintenant créer un composant visuel qui effectue une tâche et comprend au moins une propriété, une méthode et un événement. Il est assez sommaire, et son intérêt est avant tout didactique. Cela montrera aussi comment le dériver à partir du sommet de la hiérarchie des composants. Tous les composants ont TComponent dans leur arbre généalogique. TMult est un descendant direct de TComponent.

Créer TMult

Le composant TMult a deux propriétés de type integer qui peuvent être définies au moment de la conception ou de l'exécution. Il a une méthode, DoMult. Lorsque celle-ci est exécutée, les deux valeurs des propriétés sont multipliées et le résultat est placé dans une troisième propriété appelée Res. Un événement, OnTooBig est aussi implémenté. Si l'un des deux nombres est défini comme étant supérieur à 100 lorsque la méthode DoMult est appelée, le composant appelle le code vers lequel l'utilisateur à pointé dans la page d'événements. TMult est un exemple de composant purement fonctionnel : il n'a pas de composante graphique. Parmi les composants standard du même type, on peut citer TTimer, TDataBase et ceux de Table.

TMult contient les propriétés suivantes :

Val1	La première valeur à multiplier. Elle est disponible lors de la conception et de l'exécution.
Val2	La deuxième valeur à multiplier. Elle est disponible à la conception et à l'exécution.
Res	La valeur obtenue en multipliant Val1 et Val2. Disponible seulement en cours d'exécution.

TMult contient une méthode, DoMult, qui implémente la multiplication de Val1 par Val2.

L'événement de TMult, OnTooBig, appelle le Gestionnaire d'événements de l'utilisateur (s'il existe) lorsque Val1 est multipliée par Val2 et qu'une des deux valeurs est supérieure à 100.

Construire TMult

La première étape consiste à créer une unité qui constituera le noyau central de tous les composants. Pour ce faire, utilisez l'Expert composant comme précédemment et nommez le fichier Tmultiply. Pour remplir les champs de l'Expert pour créer Tmult, indiquez :

Nom de classe : Tmult
Type ancêtre : Tcomponent
Page de palette : Exemples

Le code généré comprend un noyau pour la déclaration de classe de TMult et une procédure pour enregistrer la fonction. Pour développer les propriétés, la méthode et l'événement, vous devez modifier la définition de classe et fournir les procédures et fonctions correspondantes.

Double-cliquez sur l'unité TMultiply dans le Gestionnaire de paquets pour voir apparaître le code source.

AJOUTER DES PROPRIÉTÉS

TMult contient trois propriétés : Val1, Val2 et Res. Val1 et Val2 sont disponibles lors de la conception et de l'exécution, tandis que Res ne l'est que lors de l'exécution. Comme chaque propriété contient des données, vous devez définir dans la classe TMult des variables qui contiendront ces dernières. Les utilisateurs n'accèdent aux variables que par un appel spécialisé. Par conséquent, on déclare les variables contenant les données des trois propriétés dans la partie Private de la définition de classe, ce qui signifie que seules les fonctions et les procédures de la classe pourront accéder aux données. Par convention, les noms des variables commencent par F, suivi du nom de la propriété. Dans notre exemple, toutes les propriétés sont des entiers, et leurs variables associées sont donc déclarées de type integer. La déclaration de classe est la suivante :

```
type
        TMult = class(TComponent)
        Private
                FVal1 : integer;
                FVal2 : integer;
                FRes : integer;
        Protected
        Public
        Published
end;
```

Vous devez maintenant déclarer les propriétés elles-mêmes. Utilisez le mot clé property dans la définition de classe. La définition de propriété peut apparaître généralement en deux endroits. Si une propriété est accessible lors de la conception, elle doit être déclarée dans la section Published de la déclaration. Si elle n'est disponible que lors de l'exécution, elle est placée dans la section public. Dans notre exemple, les propriétés sont stockées sous forme de types de données simples et on n'effectue pas d'action particulière lorsque les données sont lues ou écrites. On peut, par conséquent, utiliser un accès direct pour lire et écrire la propriété. Avec l'*accès direct*, vous indiquez à Delphi de modifier ou de renvoyer les données d'une variable lorsqu'une propriété est écrite ou lue. Les méthodes read et write définissent les variables.

Voici les définitions de nos trois propriétés :

```
type
        TMult = class(TComponent)
        Private
                FVal1 : integer;
                FVal2 : integer;
                FRes : integer;
        protected
        public
                Property Res:integer read FRes; {Propriété pour obtenir le résultat}
```

```
published
        property Val1:integer read FVal1 Write FVal1 default 1;
        property Val2:integer read FVal2 Write FVal2 default 1 ;
end; {TMult}
```

Comme la propriété Res est en lecture seule, vous n'avez pas besoin d'une méthode à accès direct pour écrire dans la variable FRes.

Val1 et Val2 sont définies à 1 par défaut, ce qui peut prêter à confusion. La valeur par défaut d'une propriété est en fait définie dans une autre étape de la création du composant, lorsqu'un ajoute un constructeur. Delphi l'utilise dans la ligne de propriété pour déterminer s'il doit l'enregistrer lorsqu'un utilisateur enregistre un fichier de fiche. Quand ce dernier ajoute ce composant à une fiche et laisse Val1 à 1, la valeur n'est pas enregistrée dans le fichier .dfm. Si la valeur est autre, elle est enregistrée.

Vous avez déclaré les propriétés du composant. Si vous installez le composant maintenant, Val1 et Val2 apparaîtront dans l'onglet Propriétés. Il vous reste quelques étapes avant d'obtenir un composant fonctionnel.

Ajouter le constructeur

Un constructeur est appelé lorsqu'une classe est créée. C'est lui qui est souvent chargé de l'allocation de mémoire dynamique ou de la collecte des ressources dont a besoin une classe. Il définit aussi les valeurs par défaut des variables d'une classe. Lorsqu'un composant est ajouté à une fiche, lors de la conception ou de l'exécution, le constructeur est appelé. Pour déclarer ce dernier dans la définition de classe, ajoutez une ligne constructor dans la partie public de la déclaration de classe. Par convention, on utilise Create comme nom de la procédure du constructeur, comme on peut le voir dans cet exemple :

```
{…}
public
        constructor Create(AOwner : TComponent); override;
{…}
```

On passe un paramètre au constructeur : le composant auquel appartient le constructeur. Ce n'est pas le même cas de figure que pour la propriété ancêtre. Vous devez spécifier que vous souhaitez ignorer le constructeur par défaut de la classe de l'ancêtre, qui est TComponent. Dans la portion implémentation de l'unité, vous ajoutez le code pour le constructeur.

```
constructor TMult.Create(AOwner: TComponent);
BEGIN
        inherited Create(AOwner); {appel du constructeur pour la classe du parent}
        FVal1 := 1; {Défaut pour valeur 1 }
        FVal2 := 1; {Défaut pour valeur 2 }
END; {End constructor}
```

Pour qu'une construction spécifique à un parent soit effectuée, vous devez commencer par appeler la procédure Create héritée. En l'occurrence, la seule étape supplémentaire consiste à définir des valeurs par défaut pour Val1 et Val2 qui correspondent à la section des valeurs par défaut des déclarations de propriétés.

AJOUTER UNE MÉTHODE

Une méthode est plus facile à implémenter qu'une propriété. Pour en déclarer une, placez une procédure ou une fonction dans la partie public de la définition de classe et écrivez la fonction ou procédure associée. En l'occurrence, vous ajoutez la méthode DoMult :

```
{…}
public
        procedure DoMult; {Méthode pour multiplier}
{…}

procedure TMult.DoMult;
Begin
        FRes := FVal1 * FVal2
End;
```

Votre composant fonctionne maintenant. L'utilisateur peut l'ajouter à une fiche et définir des valeurs pour Val1 et Val2 lors de la conception et de cette dernière étape. Lors de l'exécution, la méthode DoMult peut être appelée pour effectuer la multiplication.

AJOUTER UN ÉVÉNEMENT

Un événement permet à l'utilisateur d'exécuter un code spécialisé lorsque que quelque chose arrive. Pour votre composant, vous pouvez ajouter un événement qui est déclenché lorsque Val1 ou Val2 est plus grand que 100 et que DoMult est exécuté ou, par exemple, modifier le code pour que, dans une telle éventualité, Res reste inchangé.

En Delphi, un événement est une propriété spécialisée, c'est-à-dire un pointeur vers une fonction. Tout ce qui s'applique aux propriétés s'applique dans ce cas aux fonctions.

Le composant doit fournir une dernière information : le moment où il faut appeler le Gestionnaire d'événements de l'utilisateur. Rien de plus simple. Dès qu'il est temps de déclencher un événement, il suffit de voir si l'utilisateur a défini un Gestionnaire d'événements. Si c'est le cas, on appelle l'événement. Vous devez ajouter les déclarations suivantes à la définition de classe :

```
{…}
Private
        FTooBig : TNotifyEvent;
{…}
published
        Property OnTooBig:TNotifyEvent read FTooBig write FTooBig;
{…}
end; {TMult}
{…}
```

Le type TNotifyEvent est utilisé pour définir FTooBig et OnTooBig est un type de pointeur de fonction générique qui passe un paramètre de type component, Self en général. La dernière étape consiste à modifier la procédure Tmult.DoMult pour qu'elle appelle le Gestionnaire d'événements si l'un des nombres est trop grand. Avant d'appeler ce gestionnaire, vous devez regarder si un événement a été défini. Pour ce faire, utilisez la fonction assigned, qui renvoie True si un événement est défini pour le Gestionnaire d'événements et False sinon.

Le Listing suivant montre le code du composant.

Le composant TMult au complet

```
unit TMultiply;
interface

uses
SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;

type
TMult = class(TComponent)

Private
        FTooBig : TNotifyEvent;
        FVal1 : integer;
        FVal2 : integer;
        FRes : integer;
protected
public
        {constructeur principal }
        constructor Create(AOwner : TComponent); override;
        {méthode pour multiplier }
        procedure DoMult;
        {propriété pour obtenir le résultat }
        Property Res:integer read FRes;
published
        property Val1:integer read FVal1 Write FVal1 default 1; {Opérande 1}
        property Val2:integer read FVal2 Write FVal2 default 1; {Opérande 2}
        {événement}
        Property OnTooBig:TNotifyEvent read FTooBig write FTooBig;
end; {TMult}

procedure Register;

implementation

constructor TMult.Create(AOwner: TComponent);
BEGIN
        inherited Create(AOwner);
        FVal1 := 1;
        FVal2 := 1;
End;
```

```
procedure TMult.DoMult;
Begin
        if (Val1 < 100) and (Val2 < 100) then
                FRes := FVal1 * FVal2
        else
                if assigned(FTooBig) then OnTooBig(Self);
End;

procedure Register;
begin
        RegisterComponents('Samples', [TMult]);
end;
end.
```

Pour obtenir le produit de deux nombres, le composant TMult s'appuie sur un processus peu élégant en deux étapes. Lorsque la valeur de Val1 ou Val2 est modifiée, Res doit être recalculé automatiquement. Pour ce faire, vous pouvez appeler une procédure dès que la propriété est modifiée, ou encore une fonction qui renvoie la valeur de la propriété dès que celle-ci est lue.

Une méthode d'accès *est le processus d'appel d'une procédure ou d'une fonction lorsqu'on accède à une propriété. Pour en utiliser une, vous remplacez le nom de la variable de stockage direct par celui de la fonction utilisée pour manipuler les données de la déclaration de propriété.Pour implémenter une méthode d'accès dans la méthode* TMult*, vous devez apporter les modifications suivantes à la déclaration de classe :*

```
{...}
type
        TMult = class(TComponent)
Private
        FTooBig : TNotifyEvent;
        FVal1 : integer;
        FVal2 : integer;
        FRes : integer;
        {********** On déplace DoMult dans la zone Private *******}
        procedure DoMult;
        {********** On ajoute la définition de SetVal1 et SetVal2 *******}
        procedure SetVal1(InVal : Integer); {Pour définir Value1}
        procedure SetVal2(InVal : Integer); {Pour définir Value2}
protected
public
        {Propriété pour obtenir le résultat }
        Property Res:integer read FRes;
        constructor Create(AOwner : TComponent); override;
published
        {********** méthodes d'accès set *******}
        property Val1:integer read FVal1 Write SetVal1 default 1; {Operand 1}
        property Val2:integer read FVal2 Write SetVal2 default 1; {Operand 2}
        Property OnTooBig:TNotifyEvent read FTooBig write FTooBig; {Event}
end; {TMult}
```

```
{...}

procedure TMult.SetVal1(InVal : Integer);
Begin
        FVal1 := InVal;
        DoMult;
End;

procedure TMult.SetVal2(InVal : Integer);
Begin
        FVal2 := InVal;
        DoMult;
End;
{...}
end.
```

Dans le programme de test, vous n'avez plus besoin d'appeler la méthode DoMult. En fait, si vous tentez de le faire, l'application ne se compile pas car la méthode a été déplacée dans la partie privée. Les fonctionnalités demeurent inchangées pour le reste.

MODIFIER UN COMPOSANT DÉJÀ EXISTANT

TMult nous a permis d'aborder de nombreux concepts liés à la création d'un composant, mais pas de voir comment dériver un composant d'un déjà existant. L'un des grands avantages de la programmation orientée objet est qu'un objet peut être dérivé d'une classe ancêtre. Ainsi, par exemple, si vous souhaitez créer un composant qui soit un bouton vert, vous n'êtes pas obligé d'écrire tout le code qui crée un bouton et prévoit toutes les interactions possibles avec celui-ci. Un bouton générique existe déjà.

Grâce à l'héritage, vous pouvez dériver un nouvelle classe qui bénéficie de toutes les fonctionnalités de sa classe parent, auxquelles peuvent s'ajouter des améliorations ou des personnalisations.

Dans l'exemple qui suit, nous allons créer un composant appelé TButClock. Il se comporte comme un bouton, à cela près que la propriété Caption est modifiée automatiquement pour contenir l'heure courante.

Pour construire un composant à partir d'un déjà existant, vous utilisez l'Expert composant pour créer son noyau et ajouter d'éventuelles nouvelles fonctionnalités. Pour cet exemple, on utilisera un thread qui sera placé dans une boucle jusqu'à destruction du composant. Le thread est en sommeil une seconde, puis appelle une fonction de callback dans le composant pour actualiser la caption avec l'heure courante.

Le composant TbutClock

```
unit unitTBC;
interface

uses
        Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
        StdCtrls, extctrls;
type
        {un type Callback pour que le thread puisse actualiser l'horloge }
        TCaptionCallbackProc = procedure(X : String) of object;
        {Objet Thread }
        TUpdateClock = class(TThread)
private
        procedure UpdateCaption;
protected
        procedure Execute; override;
public
        UpdateClockProc : TCaptionCallbackProc;
end;

{Composant principal }
TButClock = class(TButton)
private
        { Déclarations privées}
        MainThread : TUpdateClock;
        procedure UpdateCaption(X : String);
protected
        { Déclarations protégées }
public
        { Déclarations publiques }
        constructor Create(AOwner : TComponent);override;
        destructor destroy; override;
published
        { Déclarations publiées }
end;
procedure Register;

implementation

{Cette routine appelle UpdateClockProc en lui fournissant l'heure correcte }
procedure TUpdateClock.UpdateCaption;
begin
        UpdateClockProc(TimeToStr(Now));
end;
```

```
procedure TUpdateClock.Execute;
begin
        {On boucle jusqu'à ce qu'on nous dise de terminer, puis on sort }
        while (not Terminated) do
        begin
                Synchronize(UpdateCaption);
                Sleep(1000);
        end
end;

constructor TButClock.Create(AOwner : TComponent);
begin
        inherited Create(AOwner);
        {on crée le thread en mode suspendu }
        MainThread := TUpdateClock.Create(True);
        {on définit le pointeur de callback }
        MainThread.UpdateClockProc := UpdateCaption;
        {On sort le thread du mode suspendu
        MainThread.Resume;
end;

{Appelé lors de la destruction du composant }
destructor TButClock.Destroy;
begin
        { on dit au thread de s'achever }
        MainThread.Terminate;
        {On attend la fin }
        MainThread.WaitFor;
        { Nettoyage de TButClock}
        inherited Destroy;
end;


{On modifie le Caption lorsqu'on nous dit de le faire }
procedure TButClock.UpdateCaption(X : String);
begin
        Caption := X;
end;

procedure Register;
begin
        RegisterComponents('Samples', [TButClock]);
end;
end.
```

L'analyse de ce code figure dans les parties qui suivent.

Le constructeur

Le constructeur commence par exécuter tout le code nécessaire provenant de son parent. En l'occurrence, l'objet est un descendant de TButton, il est donc important que le bouton procède aux allocations ou initialisations indispensables à son bon fonctionnement. Une fois que vous avez appelé le constructeur hérité, vous créez une instance de la classe de thread TupdateClock en appelant sa méthode create. Celle-ci est passée comme True pour indiquer que le thread doit être suspendu lors de sa création. Une fois ce dernier créé, vous devez procéder comme suit :

1. Affectez la procédure UpdateCaption à la propriété UpdateClocProc du thread. Vous indiquez ainsi au thread ce qu'il doit faire lorsqu'il est temps d'actualiser le libellé du bouton.

2. Sortez le thread du mode suspendu en appelant Resume, comme le montre le code ci-après :

```
constructor TButClock.Create(AOwner : TComponent);
begin
    inherited Create(AOwner);
    {on crée le thread en mode suspendu }
    MainThread := TUpdateClock.Create(True);
    {on définit le pointeur de callback }
    MainThread.UpdateClockProc := UpdateCaption;
    {on sort le thread du mode suspendu }
    MainThread.Resume;
end;
```

Le destructeur

Pour le destructeur, vous devez ordonner au thread de s'achever, puis attendre qu'il le soit avant de pouvoir détruire sans danger le composant TButClock, comme le montre le code ci-après :

```
destructor TButClock.Destroy;
{appelé lors de la destruction du composant }
begin
{ indique au thread de s'achever }
MainThread.Terminate;
{ on attend la fin }
MainThread.WaitFor;
{ Nettoyage de TButClock}
inherited Destroy;
end;
```

La procédure UpdateCaption

Le thread appelle UpdateCaption et lui passe Time pour actualiser le libellé, comme le montre le code ci-après (vous auriez tout aussi bien pu placer la logique liée à l'heure dans cette procédure) :

```
procedure TButClock.UpdateCaption(X : String);
{On modifie le Caption lorsqu'on nous dit de le faire }
begin
        Caption := X;
end;
```

La procédure Register

Vous achevez le composant avec la procédure Register, qui déclare que le composant TbutClock doit être placé sur la page Exemples de la VCL.

```
procedure Register;
begin
        RegisterComponents('Samples', [TButClock]);
end;
end.
```

TButClock

Une fois que vous avez construit votre composant, il est prêt à l'emploi. Lorsque vous l'ajoutez à une forme lors de la conception, il donne l'heure avant même que le programme ne soit compilé. Vous n'avez pas modifié la fonctionnalité de la propriété Caption pour qu'elle puisse être lue : elle indique l'heure courante. Vous n'avez pas empêché l'utilisateur d'écrire dans le champ Caption. Tout ce qu'il écrit dans cette propriété sera remplacé par l'heure courante la prochaine fois que le thread inscrira la nouvelle heure.

DÉCLARER UN NOUVEL ÉVÉNEMENT : USERPLOT

Lorsque vous avez travaillé avec TMult, vous avez vu comment ajouter un événement à un composant. C'était cependant un cas particulier. Vous avez déclaré la propriété comme un type TNotifyEvent. Pour passer d'autres paramètres vers ou en provenance d'un événement, Delphi a déclaré TNotify comme un pointeur vers une fonction à laquelle on transmet comme paramètre un TObject. Si un événement doit utiliser d'autres paramètres, vous pouvez déclarer le type correspondant. L'exemple qui suit montre comment y parvenir. Il permet de créer un événement qui passe un nombre réel au Gestionnaire d'événements et renvoie un nombre réel différent en utilisant un paramètre var.

Selon le type d'applications que vous concevez, le composant qui suit peut s'avérer utile. On est souvent amené à tracer une fonction mathématique. De nombreux composants vous permettent de créer un graphique en fournissant un ensemble de points, mais très peu vous

permettent de fournir simplement la fonction à tracer. Ce composant sert à cela. Un événement appelé OnUserFunc est défini. Il passe une valeur X et attend qu'une valeur Y soir renvoyée. Les facteurs d'intervalle et d'échelle sont définis comme propriétés. Ainsi, si vous souhaitez tracer la fonction $Y = X^2$, vous ajoutez le composant à votre forme et le code ci-après à l'événement OnUserFunc :

```
procedure TForm1.FuncGraph1UserFunc(X: Real; var Y: Real);
begin
        Y := X * X;
end;
```

Le composant TFuncGraph gère toutes les mises à l'échelle et transforme les coordonnées. Vous n'avez en fait à taper qu'une seule ligne de code. Les exemples précédents vous ont montré comment en implémenter la plus grande partie. Le code complet de TFuncGraph figure ci-dessous. Les sections qui suivent mettent l'accent sur la méthode permettant de créer un nouveau type d'événement et sur la création d'un événement basé sur celui-ci.

Le composant TFuncGraph

```
unit PlotChart;
interface

uses
        Windows, Messages, SysUtils, Classes, Graphics, Controls,Forms, Dialogs;

type
        TUserPlotFunc = procedure(X : real ; var Y : real) of object;
        TFuncGraph = class(TGraphicControl)
private
        { Déclarations privées }
        FRangeMinX : integer;
        FRangeMaxX : integer;
        FRangeMinY : integer;
        FRangeMaxY : integer;
        FUserFunc : TUserPlotFunc;
protected
        { Déclarations protégées }
        procedure paint; override;
public
        { Déclarations publiques }
        constructor Create(Aowner : TComponent); override;
published
        { Déclarations publiées }
        property RangeMinX : integer read FRangeMinX write FRangeMinX;
        property RangeMaxX : integer read FRangeMaxX write FRangeMaxX;
        property RangeMinY : integer read FRangeMinY write FRangeMinY;
        property RangeMaxY : integer read FRangeMaxY write FRangeMaxY;
        property OnUserFunc : TUserPlotFunc read FUserFunc write FUserFunc;
        property Width default 50;
        property Height default 50;
end;
```

```
procedure Register;
implementation

constructor TFuncGraph.Create(Aowner : TComponent);
begin
        {on définit une largeur, une hauteur et un intervalle par défaut }
        inherited Create(AOwner);
        Height := 50;
        Width := 50;
        FRangeMaxX := 1;
        FRangeMaxY := 1;
end;

procedure TFuncGraph.Paint;
var
        X,Y : integer; { pixels réels}
        RX,RY : real; {coordonnées utilisateur }
begin
        inherited Paint;
        Canvas.Rectangle(0,0,Width,Height);
        For X := 1 to Width do
        begin
                {on convertit X en X utilisateur }
                {Note : la largeur ne peut être 0}
                RX := FRangeMinX + (((FRangeMaxX - FRangeMinX)/Width)*X);

                {Si l'utilisateur a affecté une fonction de traçage }
                {on appelle cette fonction, sinon on affecte RY = 0 }
                if assigned(FUserFunc) then
                        FUserFunc(RX,RY)
                else
                        RY := 0;
                {On reconvertit RY en coordonnées de pixel }
                Y  :=  round((1-((RY-FRangeMinY)/(FRangeMaxY-FRangeMinY)))*
                Height);

                if X = 1 then
                        Canvas.MoveTo(X,Y)
                else
                        Canvas.LineTo(X,Y);
        end;
end;

procedure Register;
begin
        RegisterComponents('Additional', [TFuncGraph]);
end;

end.
```

L'analyse de ce listing figure dans les parties qui suivent.

CRÉER UN NOUVEAU TYPE D'ÉVÉNEMENT

La fonctionnalité clé de ce composant est de permettre à l'utilisateur de définir une fonction arbitraire à tracer. Pour ce faire, vous implémentez un nouveau type d'événement, TUserPlocFunc, dont la définition est :

```
TUserPlotFunc = procedure(X : real ; var Y : real) of object;
```

Ce type est déclaré dans la partie type de l'unité. Notez que TUserPlotFunc est une procédure, ce qui peut sembler bizarre dans une section type. Cela signifie que vous pouvez déclarer une variable qui est un pointeur vers une procédure prenant les arguments spécifiés dans la déclaration de type. Une fois le type déclaré, vous définissez une propriété publiée de TuserPlotFunc pour créer un événement utilisant les paramètres définis précédemment :

```
published
    property OnUserFunc : TUserPlotFunc read FUserFunc write FUserFunc;
```

Lorsque le composant est installé, un nouvel événement appelé OnUserFunc apparaît dans la liste. Si on double-clique dessus, Delphi crée une nouvelle procédure contenant les paramètres adéquats.

```
procedure TForm1.FuncGraph1UserFunc(X: Real; var Y: Real);
begin
end;
```

Appeler l'événement

Pour appeler le Gestionnaire d'événements à partir de votre composant, vous appelez la variable qui pointe sur la procédure et vous passez les paramètres adéquats. Assurez-vous qu'un événement valide est défini. Pour le savoir, appelez la fonction assigned. Voici un exemple d'appel à la fonction utilisateur :

```
if assigned(FUserFunc) then
    FUserFunc(RX,RY)
```

TFuncGraph

En ne tapant que huit lignes de code (voir ci-dessous), vous pouvez tracer quatre fonctions mathématiques.

Voilà ce qui s'appelle du développement rapide d'application.

Programme de test de TFuncGraph

```
unit unitPlotApp;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls,Forms, Dialogs,
    StdCtrls, PlotChart;

type
    TForm1 = class(TForm)
        FuncGraph4: TFuncGraph;
        FuncGraph1: TFuncGraph;
        FuncGraph2: TFuncGraph;
        FuncGraph3: TFuncGraph;
        Label1: TLabel;
        Label2: TLabel;
        Label3: TLabel;
        Label4  TLabel;
        Procedure FuncGraph1UserFunc(X: Real; var Y: Real);
        procedure FuncGraph3UserFunc(X: Real; var Y: Real);
        procedure FuncGraph4UserFunc(X: Real; var Y: Real);
        procedure FuncGraph2UserFunc(X: Real; var Y: Real);
private
public
end;
var
    Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FuncGraph1UserFunc(X: Real; var Y: Real);
begin
    Y := X;
end;

procedure TForm1.FuncGraph3UserFunc(X: Real; var Y: Real);
begin
    Y := Cos(X);
end;

procedure TForm1.FuncGraph4UserFunc(X: Real; var Y: Real);
begin
    Y := Sin(X);
end;

procedure TForm1.FuncGraph2UserFunc(X: Real; var Y: Real);
begin
    Y := sqrt(X);
end;
end.
```

Cette application de test simple trace quatre fonctions, Y=X, Y=sin(X), Y=cos(X) et Y=sqrt(X). L'échelle pour les coordonnées Y ainsi que l'intervalle des X sont définies dans les propriétés de chacune des quatre coordonnées. Les composants TLabel *sont utilisés pour inscrire un titre sous chacune des courbes.*

TYPES DE COMPOSANTS

Il existe trois types de composants :

Les **composants non visuels** (tels TTable, TDataSource , etc ...) qui ne sont en fait que des ensembles de fonctions.

Les **composants visuels susceptibles de détenir le focus**. Ces composants sont gérés par un handle par Windows et sont donc relativement gourmands en ressources.

Les **composants visuels ne pouvant pas détenir le focus** (tels TLabel ou TShape). Ce sont des dessins redessinés en permanence.

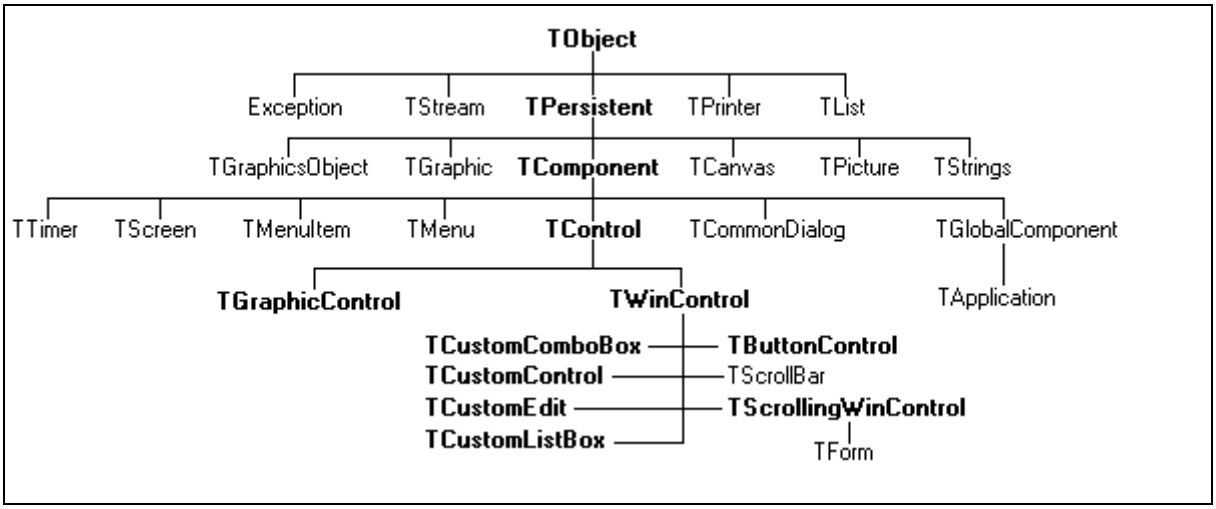
HIÉRARCHIE DES CLASSES DE COMPOSANTS

Lorsque l'on souhaite créer un composant il faut connaître parfaitement la hiérarchie des classes de composants (elle même constituant un sous-ensemble de la bibliothèque des objets de DELPHI) dans la mesure où tout composant créé doit dériver d'un composant existant.

Tous les composants dérivent de la classe générale TComponent qui possède les caractéristiques de base nécessaires (comme la possibilité de s'intégrer dans la palette des composants).

La plupart des composants de la palette dérivent de la classe TWinControl. Les composants graphiques (sans handle) dérivent quant à eux de TGraphicControl. Les composants non visuels dérivent de TComponent.

La hiérarchie "utile" est alors la suivante :



CREATION D'UNE ICÔNE SPECIFIQUE

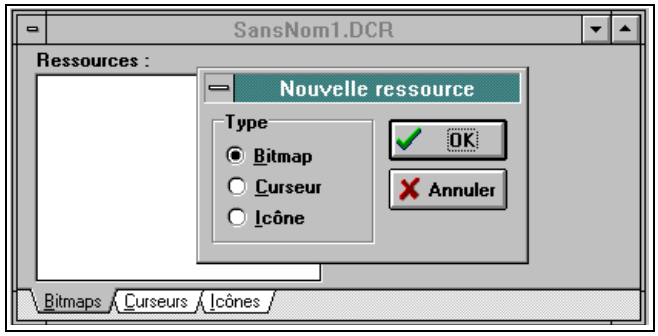
Il est possible de créer une icône spécifique en utilisant pour cela l'éditeur d'icône proposé par DELPHI. Cette icône, au format bitmap, est considérée comme une ressource par DELPHI et est stockée dans un fichier dont l'extension est .DCR.

Pour créer une icône spécifique il faut donc réaliser les opérations suivantes :

1. Ouvrir l'éditeur d'image (menu Outils | Editeur d'image).

2. Dans l'éditeur , activer le menu Fichier | Nouveau. Une boite de dialogue apparaît où il faut sélectionner l'option 'Nouveau composant (.DCR)'.

3. Une nouvelle boîte de dialogue apparaît. Elle permet d'accéder à tout type de ressource. Dans notre cas il faut activer le bouton 'Nouvelle' et indiquer qu'il va s'agir d'une ressource bitmap.



La boîte de dialogue permettant de définir une nouvelle ressource de type bitmap.

4. La validation du choix provoque l'affichage d'une nouvelle boite de dialogue dans laquelle il faut définir le nombre de couleurs utilisées et la taille de l'image à créer.

5. Même si DELPHI préconise une taille de 24 x 24 pixels, il est préférable de se limiter à une image de 20 x 20 pixels de manière à préserver les bordures de la palette de composants.

6. Au sortir de la boite de dialogue, un nom par défaut (BITMAP_1) est donné à l'icône et l'éditeur d'image est (enfin ...) affiché.

7. Créer l'image en utilisant les différents outils disponibles.

8. Il est utile d'utiliser la possibilité de zoom (X 4) proposé en haut à droite de l'éditeur.

9. Lorsque l'icône est créée il faut sauvegarder le fichier de ressources, au format DCR, en respectant les conventions suivantes :

10. Le fichier .DCR doit avoir le même nom que le fichier (.PAS ou .DCU) du composant.

11. Le fichier .DCR doit être stocké dans le même répertoire que le fichier contenant le composant.

12. Il faut enfin renommer la ressource incluse dans le fichier .DCR. Pour cela il faut :

13. Revenir à la boîte de dialogue permettant d'accéder à tous les types de ressources ;

14. Sélectionner la ressources BITMAP_1 concernée ;

15. Activer le bouton 'Renommer' afin de donner à cette ressource le même nom que la classe du composant.

16. Il ne faut utiliser que des lettres majuscules pour nommer la ressource.

17. Sauvegarder le fichier .DCR modifié.

18. Lors de la recompilation de la bibliothèque de composants, l'image est prise en compte et est affichée dans la palette de composants.

Exemple :
Si l'on a créé un composant dont la classe s'appelle TMonComposant et l'unité (et donc les fichiers .PAS et .DCU) s'appellent MonComp , il faut :

- Sauvegarder le fichier de ressources sous le nom MONCOMP.DCR;
- Nommer la ressource bitmap sous le nom TMONCOMPOSANT.

DIFFUSION D'UN COMPOSANT :

Lorsque le composant est achevé il est possible de le diffuser auprès des utilisateurs.

La diffusion peut se faire sous deux formes :

Diffusion du **fichier .PAS** du composant et du fichier .DCR associé :
Dans ce cas l'utilisateur du composant a accès à son source et peut, éventuellement le modifier. Connaissant la structure interne du composant il peut facilement réaliser un composant spécifique dérivé du composant fourni.

Diffusion du **fichier .DCU** du composant et du fichier .DCR associé :
Dans ce cas l'utilisateur peut utiliser le composant tel quel mais aura plus de difficulté à créer un composant spécifique dérivé du composant fourni du fait qu'il ne connaît pas sa structure interne.

Dans les deux cas la procédure d'installation est la même : installation du composant générant la recompilation de la bibliothèque.

LES DIFFERENTS TYPES DE PROPRIÉTÉS

Une propriété peut être d'un des types de données du Pascal ou alors être un objet. Selon les cas elles apparaîtront de différentes formes dans l'inspecteur d'objet.

Type	*Observations*
Numérique ou Chaîne	Apparaissent, sous format ASCII, dans une zone d'édition simple que l'on peut modifier directement
Enuméré	Une liste de toutes les valeurs (booléennes) que peut prendre la propriété apparaît dans l'inspecteur d'objets
Ensemble	La propriété apparaît avec un '+' sur son coté gauche. Le fait de double-cliquer sur la propriété affiche alors toutes les valeurs disponibles que l'on peut alors traiter individuellement.
Objet	On accède aux propriétés de cet objet à partir de 'inspecteur (si elles sont publiées) ou à l'aide d'un éditeur de propriétés particulier accessible en cliquant sur les '...' qui apparaissent à droite de la colonne des valeurs. Dans ce cas il faut créer l'éditeur de propriétés nécessaire.

Création d'une propriété

Pour créer une propriété il faut utiliser le mot réservé **'property'** selon la syntaxe suivante :

property *NomPropriete : type* ***read*** *[AccesLecture]* ***write*** *[AccesEcriture]* ***default*** *valeur*
où :

read [AccesLecture]	Indique quel est le mode d'accès en lecture. AccesLecture peut être une donnée membre ou une fonction de contrôle.
write [AccesEcriture]	Indique quel est le mode d'accès en écriture. AccesEcriture peut être une donnée membre ou une procédure de contrôle.
default valeur	Indique quelle est le valeur par défaut de la propriété.

Par tradition, si on ne réalise pas un accès direct à la donnée membre, la fonction de contrôle d'accès en lecture utilise le préfixe Get ... suivi du nom de la propriété et ne demande pas de paramètre. De même la procédure d'accès en écriture utilise le préfixe Set .... et demande un paramètre d'un type permettant de modifier la donnée cible.

Seuls le nom de la propriété et son type sont obligatoires. Toutes les autres spécifications sont facultatives (en particulier la clause 'default').

Cependant si on trouve souvent des propriétés qui ne sont accessibles qu'en lecture (pas de clause 'write') il n'existe pratiquement pas de propriété qui ne sont accessibles qu'en écriture (pas de clauses 'read').

Le fait d'assigner une valeur par défaut à une propriété ne signifie pas qu'elle prendra cette valeur lors de la création du composant. Cela signifie simplement que si la valeur de la propriété a cette valeur lors de la conception d'une feuille, au sein d'un projet, il n'y aura pas de ligne référençant cette propriété dans le fichier .DFM associé. D'où économie lors de la génération du module objet (et donc de l'exécutable) associé.

Par contre il est souhaitable d'initialiser la propriété à sa valeur par défaut lors de l'exécution du constructeur du composant.

Publication d'une propriété

Pour qu'une propriété apparaisse dans l'inspecteur d'objet il faut que sa définition soit déclarée dans la section 'published' de la classe.

Il faut aussi que la propriété définisse une clause 'read' sinon elle n'apparaîtra pas dans l'inspecteur d'objet.

Les propriétés qui sont déclarées dans la section 'public' ne sont accessibles qu'à l'exécution.

Une propriété peut par ailleurs être définie dans la partie 'public' ou 'protected' de la classe. Dans ce cas elle n'apparaîtra pas dans l'inspecteur d'objet mais une classe dérivée à partir de cette classe pourra publier la propriété.

C'est sur ce principe que sont réalisées les classes des composants standards. Elles dérivent toutes d'une classe de plus haut niveau, de préfixe TCustom..., dans laquelle les propriétés ne sont pas publiées. La classe du composant standard se contente de publier les propriétés désirées (certaines propriétés pouvant rester non publiées).

Le composant TListbox dérive ainsi de la classe TCustomListBox.

Pour publier une propriété, déclarée 'protected' dans le classe mère, il suffit de l'invoquer par son nom dans la section 'published' :

```
published
    Color ;{ rend publiée une propriété définie dans la classe mère }
    ....;
```

Exemples de propriétés :

```
type
    TCouleur = ( clRed, clWhite, clBlue, clGreen, clBlack ) ;
    { Définition d'un type énuméré }

    TMaClasse = class ( TComponent )
    private
        FLongueur : integer ;
        FChaine : string ;
        FCouleur : TCouleur ;
        function GetCouleur : TCouleur ;
        procedure SetCouleur ( couleur : TCouleur ) ;
```

```
published
        property Longueur: integer read FLongueur write FLongueur
                default 20 ;
        property Chaine : string read FChaine write FChaine ;
        property Couleur : TCouleur read GetCouleur write SetCouleur ;
end ;

......

function TMaClasse.GetCouleur : integer ;
begin
        result := FCouleur ;
end ;

procedure TMaClasse.SetCouleur ( couleur : integer ) ;
begin
        FCouleur := couleur ;
end ;
```

CONSTRUCTION ET DESTRUCTION D'UN COMPOSANT

Comme pour tout objet, la définition d'un composant implique celle d'un constructeur et d'un destructeur.

Propriétaire d'un composant

La différence essentielle qui existe entre un composant et un objet vient du fait qu'un composant doit toujours appartenir à un autre composant (normalement au composant de type TForm dans le cas d'une application traditionnelle).

La détermination du propriétaire du composant se fait lors de l'exécution du constructeur de ce dernier : il est passé en paramètre au constructeur.

Si on ne veut pas qu'un autre composant soit propriétaire d'un composant créé, il suffit de transmettre Self (qui correspond à la fiche) comme valeur du paramètre AOwner.

Le propriétaire du composant est chargé de le détruire le cas échéant. En particulier lorsque le propriétaire est détruit (par exemple lorsque la feuille est détruite) c'est lui qui, automatiquement, détruit les composants dont il est propriétaire en appelant pour chacun d'entre eux leur méthode Free.

On peut connaître le parent d'un composant en invoquant la propriété Owner.

Parent d'un composant

Dans le cas d'un composant visuel il faut en outre définir un composant parent. Celui-ci est le composant dans lequel le composant créé va être affiché.

Seuls les composants "containers" peuvent être parents d'autres composants (ex : TPanel, TGroupBox, etc ....).

Le propriétaire et le parent peuvent être différents.

Par exemple le propriétaire d'un composant peut (c'est le cas général) être la feuille alors que son parent peut être un composant Panel.

La détermination du composant parent se fait lors de l'exécution du constructeur du composant en initialisant sa propriété Parent.

```
Parent := self { le parent est le propriétaire }
ou
Parent := PAffiche { le parent est un composant de type Panel }
```

A partir du moment où on a défini un parent, les coordonnées (propriétés Top et Left) du composant créé sont définies à partir du coin haut / gauche du composant parent.

LES ÉVÉNEMENTS

La deuxième caractéristique majeure d'un composant est son aptitude à réagir à des événements.

En fait, lorsque l'on programme normalement en DELPHI, on ne s'occupe plus particulièrement que de définir quelle est la réaction de l'application à un événement intervenant sur un composant particulier (création de gestionnaire d'événement).

C'est ce qui se traduit par un code du type :

```
procedure TForm1.Button1Click (Sender: TObject);
begin
      .....          { code définissant l'action à exécuter lorsque l'événement
Click se                    produit sur le composant Button1 }
end ;
```

A ce niveau il y aura lieu de distinguer :

- Les événements standards, déjà pris en compte par la classe mère du composant ;
- Ceux dont il serait souhaitable de modifier le comportement par défaut ;
- Ceux qu'il faudra créer de toute pièce.

Il faudra surtout prendre en compte que si le concepteur de composant met à la disposition de l'utilisateur un certain nombre d'événements gérés, c'est l'utilisateur qui crée le gestionnaire d'événement associé. Le concepteur ne peut donc faire aucune présupposition sur l'action du gestionnaire d'événement. Il ne doit pas, en outre, empêcher, par une modification interne au composant, l'action du gestionnaire d'événement.

En première analyse, sauf à créer un composant de toute pièce, la plupart des composants standards définis dans les classes de base ont un comportement par défaut satisfaisant. Par ailleurs les principaux cas (correspondant aux normes d'ergonomie de Windows) sont pris en compte, on n'a donc que rarement l'occasion de souhaiter gérer un nouveau type d'événement.

Evénements standards :
La classe TControl définit le comportement d'un certain nombre d'événements, dits événements standards. Ces événements sont :

OnClick,	***OnDblClick***	***OnDragDrop***	***OnDragOver***
OnEndDrag	***OnMouseDown***	***OnMouseMove***	***OnMouseUp***

Par ailleurs, la classe TWinControl définit, quant à elle, le comportement d'autres événements qui peuvent être assimilés à des événements standards. Ce sont :

OnEnter	***OnExit***	***OnKeyDown***	***OnKeyPress***	***OnKeyUp***

Caractéristiques générales d'un événement

Un événement est un **pointeur** sur une **méthode** spécifique appartenant à une **instance de composant** donnée.

Du point de vue de l'utilisateur d'un composant, un événement est un nom associé à une méthode spécifique.

L'événement OnClick du composant Button1 est associé à la procédure Button1Click qui constitue le gestionnaire d'événement.

Lorsqu'une occurrence de l'événement 'click souris' se produit sur le composant Button1 c'est le gestionnaire d'événement Button1Click qui est exécuté.

Pour le concepteur de composant il va s'agir de fournir à l'utilisateur un moyen pour rattacher son code (le gestionnaire d'événement) afin de le rendre en mesure de répondre aux occurrences d'événement.

Il ne faut pas perdre de vue que, dans tous les cas, un gestionnaire d'événement est facultatif. Le concepteur prévoit donc les événements qui seront gérés par le composant mais le code de ce dernier ne doit en aucun cas dépendre de la présence ou non d'un gestionnaire d'événement associé à un événement pour s'exécuter.

Implémentation sous DELPHI

Sous DELPHI, chaque événement géré par le composant (mis à la disposition de l'utilisateur pour qu'il puisse y rattacher le code d'un gestionnaire d'événement) est accessible via une propriété spéciale servant d'interface d'accès (en lecture et en écriture) à une donnée d'un type particulier susceptible de pointer sur le gestionnaire d'événement "utilisateur" .

La donnée membre - privée - permettant d'accéder à un - éventuel - gestionnaire d'événement, est d'un type particulier. Il s'agit d'un pointeur sur une méthode car il doit être capable de pointer sur le code de la méthode correspondant au gestionnaire d'événement créé, lors de la conception d'un projet, par l'utilisateur du composant.

La donnée membre étant un pointeur sur une méthode (toujours de type procédure), son type doit être indiqué précisément. La syntaxe générale de la déclaration d'un tel type est:

```
type
    TPointeurMethode = procedure (paramètres de l'événement ) of object ;
```

Les différents paramètres doivent être conformes à ceux de la méthode pointée (même type, même nombre, même ordre).

DELPHI propose, en standard, un certain nombre de type de "pointeurs sur une méthode", correspondant aux événements standards gérés.

TNotifyEvent : Utilisé pour les événements qui n'ont pas besoin de paramètre (ex:OnClick);

TKeyEvent : Utilisé pour les événements liés au clavier OnKeyDown et OnKeyUp ;

TKeyPress : Idem pour l'événement OnKeyPress ;

TMouseEvent : Utilisé pour les événements liés à la souris OnMouseDown et OnMouseUp;

TMouseMoveEvent :Utilisé pour les événements liés aux déplacements de la souris (OnMouseMove).

Voir l'aide en ligne pour connaître le prototype de chacun de ces types de pointeur sur méthode.

La propriété "gestionnaire d'événement"

Pour accéder à la donnée membre, privée, il faut créer une propriété publiée qui doit avoir les caractéristiques suivantes:

- Elle est de type "pointeur sur une méthode" (le même type que celui de la donnée membre associée) ;

- Elle n'utilise pas de méthode pour implémenter les parties 'read' et 'write'. Elle accède directement à la donnée privée.

Pour gérer un événement 'clic souris' on a alors le code :

```
type
    TControl = class ( TComponent )
    private
        FOnClick : TNotifyEvent ;   { définition du pointeur de méthode }
        .......
    published
        property OnClick : TNotifyEvent read FOnClick write FOnClick;
        ......
    end ;
```

Le seul fait que la propriété soit de type "pointeur sur une méthode" fait qu'elle est insérée dans le volet "événements" de l'inspecteur d'objet.

Il est conseillé de se conformer aux règles de nommage DELPHI qui préconisent qu'une propriété événement doit avoir un nom commençant par 'On' suivi du nom de la méthode de gestion de l'événement dans la mesure où au niveau de l'utilisateur, le gestionnaire d'événement associé aura un nom constitué par la concaténation du nom du composant et de celui de la méthode.

Si l'événement à gérer l'est déjà dans la classe de base, il n'y a pas lieu de recréer une nouvelle propriété ni même de définir un pointeur sur une méthode. Il suffit de publier celle du composant d'origine (dans le cas où cela n'a pas déjà été fait) selon la syntaxe habituellement utilisée pour les autres propriétés :

```
published
    property OnClick ;
```

{ la propriété est publiée et donc l'événement OnClick est accessible dans l'inspecteur d'objet pour que l'utilisateur puisse créer le code du gestionnaire d'événement }

La méthode de gestion de l'événement

Le comportement de base de chaque composant à l'occurrence d'un événement est défini dans une méthode spécifique : chaque événement standard dispose donc d'une méthode de gestion appropriée définie dans la classe TControl ou TWinControl.

Par exemple il existe une méthode KeyDown, définie dans l'unité TWinControl pour assurer la gestion d'un appui sur une touche du clavier avec récupération, dans la variable Key, de la valeur de la touche utilisée.

De même il existe une méthode Click pour réagir aux événements OnClick, une méthode MouseMove pour gérer l'événement OnMouseMove, etc.....

Par contre la méthode associée à l'événement OnExit s'appelle DoExit.

Toutes sont méthodes sont protégées et sont virtuelles. Il est donc possible de les redéclarer dans le composant à créer si l'on souhaite modifier leur comportement par défaut.

Comment ça marche ?

La gestion des événements met en oeuvre conjointement des mécanismes au niveau de Windows et au niveau de l'exécutable réalisé avec DELPHI. Dans ce dernier les différents éléments conçus par le programmeur sont présent mais il est difficile, à la simple lecture du code, de comprendre comment ils coopèrent entre-eux. Ceci pour la bonne raison qu'une partie des liens entre les éléments est réalisée directement par Windows selon le cheminement suivant et une autre selon un mécanisme interne mis en place, dans l'exécutable, par DELPHI:

1. Lorsqu'un message est généré par Windows il est diffusé vers l'application active.

2. Si le composant - actif à se moment là - contient une donnée membre de type pointeur sur une méthode (c'est à dire du type TNotifyEvent ou de ses dérivés) compatible avec ce message, le mécanisme mis en place par DELPHI cherche, dans la table des méthodes du composant, celle qui a la même "signature" (même nombre et même types de paramètres) que la donnée pointeur sur méthode et l'exécute.

3. Au sein de cette méthode il y a test de la propriété On..... correspondant au type de message. Si la valeur de cette propriété est différente de Nil c'est que l'utilisateur final a écrit un gestionnaire d'événement. Celui-ci est alors exécuté.

4. Le code particulier de la méthode de gestion de l'événement est exécuté (avant ou après celui de l'utilisateur selon les cas).

S'il n'y a pas de méthode à la signature conforme, la recherche est faite dans les classes ascendantes.

Exemple :
Si l'on souhaite redéfinir, dans un composant dérivé de TEdit, la gestion par défaut de l'événement OnKeyDown de manière à ce qu'il élimine les espaces entrés au clavier il faut écrire le code suivant :

```
type
        TMonEdit = class ( TCustomEdit )
        private
                .....
        protected
                .....
                 procedure KeyPress ( var Key: Char ) ; override ;
                .......
        published
                .....
                OnKeyPress ;
                { publication de la propriété pour la rendre accessible à l'utilisateur }
                ......
        end ;

        procedure TMonEdit.KeyPress (var Key: char );
        begin
                if Key = ' ' then
                 Key := chr ( 0 ) ;
                inherited KeyPress ;
        end ;
```

Il est obligatoire, dans le code de la méthode de gestion de l'événement de faire appel au gestionnaire par défaut (via l'instruction inherited.....) de manière à ce que le mécanisme d'appel du gestionnaire d'événement susceptible d'être mis en place par l'utilisateur du composant soit appelé et exécuté..... l'autre possibilité étant de gérer soit même cet appel, ce qui est plus délicat et peut s'avérer insuffisant (la méthode de gestion par défaut réalise peut être certaines autres fonctionnalités, comme des rafraîchissements écrans).

Dans certains cas, l'appel à la méthode originelle sera fait avant l'exécution du code assurant la modification du comportement par défaut. Dans d'autres, elle se fera après. C'est au concepteur de composant de choisir le déroulement le plus conforme à ses buts.

Il n'est pas nécessaire de définir des pointeurs sur les méthodes (ex : FOnKeyPress: TKeyPressEvent), dans la partie privée de la classe, dans la mesure où ces pointeurs ont déjà été définis dans la classe de base et son accessible via la propriété On.....

Le concepteur défini ses propres événements

Dans ce dernier cas, le plus rare, le concepteur doit définir intégralement le mécanisme permettant de réaliser un événement proposé pour utilisation à un utilisateur.

Pour cela il doit :

- Déterminer ce qui déclenche l'événement ;
- Définir le type de gestionnaire ;
- Déclarer l'événement ;
- Appeler l'événement.

Comme certaines de ces actions nécessitent des connaissances approfondies de la gestion des messages Windows, nous nous bornerons à indiquer brièvement les actions à entreprendre.

Détermination du message déclenchant l'événement

Il faut déterminer, dans la liste des messages Windows, lequel sera celui pris en compte comme déclencheur d'événement.

Pour cela il faut aller dans l'API Windows, rubrique 'messages (3.1)' afin de prendre connaissance de la liste des messages gérés par Windows.

Ces messages sont référencés sous la forme XX_YYY..... où XX indique à quel type d'événement le message s'applique (BM : message vers un bouton, CB : vers une combo-box, etc .....) ;

Il faut ensuite connaître les différentes structures de messages définies par DELPHI pour prendre en compte, en les encapsulant, les messages Windows (voir rubrique 'messages' de l'aide).

Ces types correspondent à la définition de structures. Ils sont tous nommés TWM..... .

Définition du type d'événement

Il faut alors déterminer le type de gestionnaire d'événement afin de déclarer un pointeur sur la méthode de gestion ayant un prototype adapté.

Pour cela on peut utiliser les types de pointeurs définis par DELPHI (TNotifyEvent et autres) ou alors créer sont propre type.

Déclaration de l'événement :

On peut alors déclarer la propriété pour l'événement, au type du pointeur sur une méthode déclaré précédemment.

Il faut respecter les règles de nommage afin que l'événement apparaissent dans l'inspecteur d'objet avec un nom qui soit dans la lignée des autres événements.

Appel du gestionnaire d'événement :
Il faut enfin créer le code du gestionnaire d'événement, écrit par l'utilisateur du composant, dans le cas où ce code existe.

En général le code d'appel à la forme suivante (cas d'un gestionnaire associé à l'événement OnClick) :

```
procedure TMonComposant.Click ;
begin
        If Assigned ( FOnClick ) then
                FOnClick ( self ) ;
        .......                 { code spécifique mis en place par le concepteur }
end ;
```

La méthode Assigned vérifie la valeur de la donnée FOnClick. Si cette donnée, de type pointeur sur méthode, a une valeur différente de Nil c'est que l'utilisateur du composant a créé un gestionnaire d'événement sur cet événement. Dans ce cas, le gestionnaire d'événement est appelé et exécuté, via la référence à FOnClick

Il se pose la question, comme dans un cas précédent, de savoir si le code spécifique prévu par le concepteur doit être exécuté avant ou après celui mis en place par l'utilisateur. Dans tous les cas :

- L'absence de "code utilisateur " ne doit pas empêcher l'exécution du code prévu par le concepteur (d'où le test d'existence par Assigned ( )) .
- Le code prévu par le concepteur ne doit pas influencer le code mis en place par l'utilisateur.

Exemple :
Voici le code utilisé par TControl pour gérer le message WM_LBUTTONDOWN (gestion des clics sur le bouton gauche de la souris).

```
type
        TControl = class ( Tcomponent )
        private
                FOnMouseDown : TMouseEvent;
                { définition du pointeur sur méthode : il utilise un type prédéfini }
                procedure DoMousedown ( var message : TWXMouse ;
                        button : TMouseButton ; shift :TShifstate ) ;
                { ce prototype correspond au type TMouseEvent }
                procedure WMLButtonDown ( var Message : TWMLButtonDown ) ;
                        -> message WM_LBUTTONDOWN ;
        protected
                procedure MouseDown ( button : TMouseButton ; shift :TShiftState ;
                        x , y : Integer ) ; dynamic ;
        end ;
```

```
procedure TControl.MouseDown ( button : TMouseButton ;
        shift : TShiftState ; x , y : integer ) ;
begin
        if Assigned ( FOnMousedown ) then
                FOnMousedown ( self , shift , x , y ) ;
                { Appel du gestionnaire d'événement défini par l'utilisateur }
end ;

procedure TControl.DoMousedown ( var Message : TWMMouse ;
        button : TMouseButton ; shift : TShiftState ) ;
begin
        with Message do
                MouseDown ( button , KetToShiftState ( Keys ) + Shift , XPos,
                        YPos ;
         { Appel de la méthode dynamique de gestion par défaut }
end ;

procedure TControl.WMLButtonDown (var Message :
TWMLButtonDown );
begin
        inherited ;     { exécute la gestion par défaut }
        if csCaptureMouse in ControlStyle then
                MouseCapture := True ;
        if csClickEvents in ControlStyle then
                Include ( FControlState , csClicked ) ;
        DoMousedown ( Message , mbLeft, [ ] ) ;
        { appel de la méthode spécifique au contrôle }
end ;
```

Dans une classe dérivée de TControl il faudra définir une propriété publiée OnMousedown permettant d'accéder à FOnMouseDown.

Cas particulier des événements qui n'utilisent pas les messages Windows

Il peut arriver que l'événement créé ne s'appuie pas sur le mécanisme de message propre à Windows (par exemple lorsqu'une condition vient à être satisfaite au sein du code }.

Dans ce cas une grosse partie de la difficulté à gérer de bout en bout un événement est levée car on en est réduit à mettre en place un mécanisme purement algorithmique.

Les actions qui restent à réaliser sont alors :

1. Déclaration d'un pointeur sur une méthode adéquat (il s'agira la plupart du temps d'un pointeur sur une méthode ne nécessitant pas de paramètre, donc de type TNotifyEvent)

2. Déclaration de la propriété associée On..... .

3. Création de la méthode protégée de gestion de l'événement. Cette méthode, comme dans le cas précédent, doit tester la valeur de la donnée pointeur pour, le cas échéant, exécuter le gestionnaire d'événement créé par l'utilisateur.

```
if Assigned ( FOnNonEvenement ) then
    FOnMonEvenement ( self ) ;
```

4. Appeler cette méthode à partir des différentes procédures et fonctions du composant qui peuvent, lors de leur exécution, "déclencher l'événement".

LES COMMUNICATIONS SOUS TCP/IP

DELPHI permet de réaliser des interfaces graphiques, conformes aux normes Windows.

Cependant, de plus en plus, les utilisateurs ont besoins de s'échanger des informations ou de communiquer entre eux. Jusqu'à présent, Windows fournissait un certain nombre d'outils de communications, mais ceux-ci ne correspondent pas forcement à votre besoin.

Dans ce chapitre nous allons, en nous appuyant sur TCP/IP , voir les principes de la programmation système sous Windows et utiliser un composant orienté Communication sous **DELPHI**.

Principes et outils de base ou ce qu'il faut savoir sur Windows avant d'utiliser les sockets

Le cauchemar des administrateurs (Informatiques N°23 de décembre 1996)

" Les winsock 1.1 se présentent dans la pratique sous la forme d'un fichier .DLL (Dynamic Link Library). Ce fichier s'appelle Winsock.dll (16 bits) ou wsock32.dll (32 bits). Certains aspects des spécifications Winsock 1.1 laissant la place à l'interprétation, un grand nombre d'applications fondées sur les winsock nécessitent un fichier winsock.dll particulier. Powerpoint de Microsoft offre par exemple la possibilité d'envoyer une présentation à plusieurs machines du réseau. Cette possibilité d'utilisation du mode Broadcast n'est pas incluse dans les spécifications winsock 1.1. Il s'agit d'une des nombreuses extensions apportées par Microsoft et implémentées dans les winsock fournies avec la pile TCP/IP Microsoft. L'utilisation de cette fonction avec les winsock d'un autre éditeur peut avoir des conséquences imprévisibles..."

Historique

Permettre la communication entre des postes de travail Windows et des serveurs UNIX : telle est la raison d'être de l'API (Application Programmer Interface) Windows Sockets, connue également sous l'appellation winsock.

A la fin des années 1980, les premiers postes Windows connectés sur un réseau local accèdent principalement à des serveurs de fichiers et des imprimantes NETWARE. Pour y parvenir, ils utilisent le protocole IPX/SPX de Novell. Avec l'arrivée en force de l'architecture client - serveur, le PC est élu client par excellence du nouveau modèle. Son faible coût permet de le diffuser en masse, tandis que l'interface graphique séduit les utilisateurs. Intel et Microsoft prennent rapidement du poids en greffant également leurs produits aux serveurs sous UNIX. Pour cela, ils définissent un canal de communication s'appuyant sur TCP/IP, le protocole réseau utilisé par UNIX. Ainsi naît l'API Winsock. Empruntant les souliers d'UNIX, Windows trouve enfin une voie de normalisation des échanges entre les applications et la couche TCP/IP.
Les spécifications des winsock ayant fait l'unanimité, ils représentent aujourd'hui un standard incontesté. La croissance exponentielle de l'Internet trouve sa source dans les winsock, qui transforme des dizaines de millions de plates-formes Windows en autant de clients potentiels du réseau des réseaux.

UTILISATION DES SOCKETS SOUS WINDOWS

L'API WINDOWS ne dispose pas en standard des fonctions et procédures nécessaires pour utiliser les possibilités d'un protocole lors du développement d'une application.

Pour accéder au monde des communications, il faut installer une pile TCP/IP et s'appuyer sur les fonctions proposées par une DLL spécifique :WINSOCK.DLL.

PRÉSENTATION DES WINSOCK

Spécifications
Sur le modèle des API pour socket BSD, mais pas 100 % compatibles,
Pas de licence nécessaire pour créer des applications utilisant winsock,
A l'origine, conçus pour Windows 3.11, a suivi l'évolution.
Version 1.1 : limitée à l'implantation de TCP/IP,
Version 2.0 : supporte un grand nombre d'autres protocoles
La version 32 bits est intégrée à Windows 95
Fournis sous la forme d'une librairie à liens dynamiques (DLL).

EXTENSION DES WINSOCK

La norme sous UNIX est d'émettre avec des appels bloquants (attente de connexion), ce qui n'est pas compatible avec les applications WINDOWS.
Entraîne un blocage au niveau de l'interface utilisateur,
Empêche le multitâche "coopératif" de Windows 3.11.

Les winsock supportent la notification asynchrone (par un message Windows) à la fin d'une connexion, d'une E/S socket,...

Les Winsock offrent des extensions au sockets pour les rendre plus adaptées au mode d'envoi de message entre applications Windows

PHILOSOPHIE WINSOCK

Winsock 1.1 a été conçu pour Windows 16 bits
multitâche coopératif (pas de thread)

Les opérations bloquantes sont supportées mais pas recommandées

- Problème de portage de code :

porter un code bloquant traditionnel sous Windows est facile, **mais pas recommandé**
➔adapter un code bloquant au modèle de programmation Windows est **difficile**

UTILISATION DE LA WINSOCK

Avant d'utiliser les différentes fonctions de la winsock.dll, il est nécessaire de l'initialiser.
Cela se fait en utilisant la fonction ***WSAStartup($101, info)*** avec :

info:WSADATA;
$101: version de la dll (word) ➔correspond à la version 1.1 de la winsock

00	01	00	01

le deuxième argument retourne des informations sur le mode de transport

```
WSAData = record
            wVersion : word;       {numéro de version de la DLL utilisée}
            wHighVersion:word;  {plus haut numéro de la version supporté}
            .....
            iMaxSockets:u_short; {nombre de sockets disponibles}
            iMaxUdpDg:u_short; {taille maximum d'un datagramme}
            end;
```

Pour utiliser les fonctions de la DLL, il est nécessaire de les déclarer dans la partie implémentation sous la forme :

```
implémentation
    function listen ( s:TSocket; nb:integer) : integer; stdcall; external 'WINSOCK' ;
```

La directive "**external**" permet de transmettre le nom de la DLL qui contient le code de la fonction. Le nom de la dll doit être écrit avec son extension.

LES ERREURS

Winsock définit 2 types d'erreurs pour les API socket :

➔pour les API qui retournent un socket
#define INVALID_SOCKET $FFFF

➔pour les autres API socket
#define SOCKET_ERROR (-1)

PARTICULARITÉS DE LA WINSOCK

Les applications utilisant la winsock.dll doivent appeler **WSAStartup()** avant d'utiliser les fonctions de la winsock et peuvent appeler **WSACleanup()** pour ne plus utiliser la winsock..

LES FONCTIONS DE LA WINSOCK.DLL

On retrouve dans la winsock.dll les mêmes fonctions que celles vu en programmation système. Le tableau ci-dessous un aperçu de ces fonctions. Ce référer au fichier ***socket.pas*** pour avoir la liste complète des fonctions de la DLL.

function **accept**(s: TSocket; var addr: sockaddr_in; var addrlen: integer) : TSocket;
far; external 'WINSOCK';
function **bind**(s: TSocket; var addr: sockaddr_in; namelen: integer) : integer;
far; external 'WINSOCK';
function **closesocket**(s: TSocket) : integer;far; external 'WINSOCK';
function **connect**(s: TSocket; var name: sockaddr_in; namelen: integer) : integer;
far; external 'WINSOCK';
function **htonl**(hostlong: u_long) : u_long; far; external 'WINSOCK';
function **htons**(hostshort: u_short) : u_short; far; external 'WINSOCK';
function **inet_addr**(cp: PChar) : u_long; far; external 'WINSOCK';
function **inet_ntoa**(sin: in_addr) : PChar; far; external 'WINSOCK';
function **listen**(s: TSocket; backlog: integer) : integer;far; external 'WINSOCK';
function **ntohl**(netlong: u_long) : u_long; far; external 'WINSOCK';
function **ntohs**(netshort: u_short) : u_short; far; external 'WINSOCK';
function **recv**(s: TSocket; buf: PChar; len: integer; flags: integer) : integer;
far; external 'WINSOCK';
function **recvfrom**(s: TSocket; buf: PChar; len: integer; flags: integer;
var from: sockaddr_in; var fromlen: integer) : integer; far; external 'WINSOCK';
function **send**(s: TSocket; buf: PChar; len: integer; flags: integer) : integer;
far; external 'WINSOCK';
function **sendto**(s: TSocket; buf: PChar; len: integer; flags: integer;
var saddrto: sockaddr_in; tolen: integer) : integer; far; external 'WINSOCK';
function **shutdown**(s: TSocket; how: integer) : integer; far; external 'WINSOCK';
function **socket**(af: integer; stype: integer; protocol: integer) : TSocket; far; external 'WINSOCK';
function **gethostbyaddr**(addr: PChar; len: integer; stype: integer) : phostent;
far; external 'WINSOCK';
function **gethostbyname**(name: PChar) : phostent; far; external 'WINSOCK';
function **gethostname**(name: PChar) : integer; far; external 'WINSOCK';

function **WSAStartup**(wVersionRequired: word; var lpWSAData: WSADATA) : integer;
far; external 'WINSOCK';
function **WSACleanup** : integer; far; external 'WINSOCK';
function **WSAAsyncSelect**(s: TSocket; handle: HWND; wMsg: u_int; lEvent: longint):
integer; far; external 'WINSOCK';
Toutes ces fonctions sont utilisables sous Windows et permettent de faire, dans un premier temps, de la programmation structurée comme sous UNIX.

Pour que les différents types spécifiques aux sockets soient reconnus par votre application, il suffit de rajouter ***Sockets*** dans la ligne ***uses***.

exemple : Récupération de l'heure sur un serveur distant en mode connecté ou non.

UTILISATION DE L'OBJET SOCKETS

Utilisable sous DELPHI, ce nouveau composant prend en charge la gestion des événements, ce qui permet de l'intégrer à toute application.

Les contraintes de l'objet SOCKET

✓Cet objet est prévu pour utiliser des sockets en **mode connecté**.

De nombreuses fonctions de l'API Winsock ne sont pas implémentées directement par l'objet et il est donc nécessaire de réécrire certaines fonctions lorsqu'on a des besoins spécifiques.

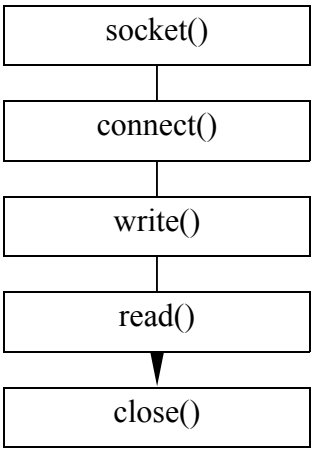
Du structuré à l'événementiel

Sous UNIX, la méthode de programmation utilisée pour créer et utiliser les sockets était structurée. Sous DELPHI, la programmation se fait en mode événementiel.

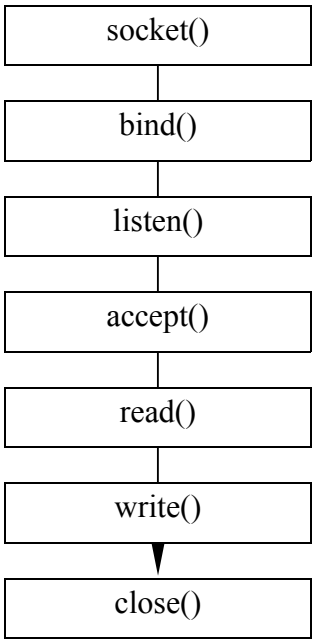
Essayons, dans un premier temps, de redéfinir les méthodes, les événements et les propriétés que devrait comporter notre composant pour fonctionner.

RAPPEL :

Client UNIX



Serveur UNIX



Découpage en Méthodes

méthode SConnect : connexion à un serveur — socket() connect()

méthode SListen : écoute sur un port — socket() bind() listen()

méthode SAccept : accepte la connexion d'un client — accept()

méthode SReceive : réception de données — read()

méthode SSend : envoi de données — write()

méthode SClose : clôture d'un socket — close()

méthode SCancelListen : arrête l'écoute sur le socket

méthode GetPort : renvoie le port du socket local

méthode GetIIAddr : renvoie l'adresse IP locale

méthode GetPeerPort : renvoie le port utilisé par le correspondant

méthode GetPeerIPAddr : renvoie l'adresse IP du correspondant

Les Evénements :

en cas d'erreur sur le socket : (client et serveur)	***OnErrorOccured***
en cas de demande de connexion : (serveur)	***OnSessionAvailable*** on y trouve en principe **SAccept**
en cas d'acceptation de connexion : (client)	***OnSessionConnected*** on y trouve le début des échanges , **SSend**
en cas de fermeture de socket : (client ou serveur)	***OnSessionClosed*** quand le correspondant a fermé le socket on y trouve en principe **SClose**
en cas de données présentes : (client et serveur)	***OnDataAvailable*** on y trouve en principe **SReceive**

Les propriétés

IPAddr : contient l'adresse IP du correspondant chez qui vous voulez vous connecter (client) ou l'adresse locale (serveur)

Port : contient le numéro du port sur lequel vous voulez vous connecter (client)ou vous mettre en écoute (serveur)

SocketNumber : retourne le numéro du socket de la connexion courante

MasterSocket : retourne le numéro du socket d'écoute

Text : permet l'envoi et la réception de texte via le socket

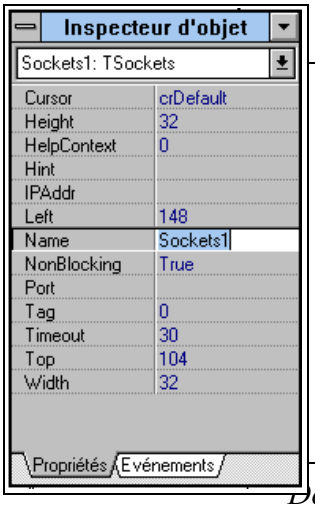
OOB : envoi les données en urgent

NonBlocking : à False pour mode bloquant
à True pour mode non-bloquant (par défaut)

TimeOut : en mode bloquant, indique le temps d'attente du socket
après ce temps, il y a génération d'une erreur

PROPRIÉTÉS DE L'OBJET SOCKET ET CARACTÉRISTIQUES DE LA WINSOCK.DLL

Détail des propriétés implémentées



Property Name	Writable	Readable	Design time	Run time
IPAddr	yes	yes	yes	yes
Port	yes	yes	yes	yes
SocketNumber	yes	yes	no	yes
MasterSocket	yes	yes	no	yes
Text	yes	yes	no	yes
Peek	no	yes	no	yes
OOB	yes	yes	no	yes
NonBlocking	yes	yes	yes	yes
Timeout	yes	yes	yes	yes

Design time : dans l'inspecteur d'objet Run time : à l'exécution

IPAddr :
contient l'adresse IP du correspondant chez qui vous voulez vous connecter.
exemple :

```
Sockets1.IPAddr:='ux150';
Sockets1.IPAddr:='208.3.3.150';
addr:=Sockets1.IPAddr;
```

Port :
contient le numéro du port sur lequel vous voulez vous connecter (client)ou vous mettre en écoute (serveur).
exemple :

```
Sockets1.Port:='7';
port:=Sockets1.Port;
```

SocketNumber :
retourne le numéro de la socket de la connexion courante
exemple :

```
sock:=Sockets1.SocketNumber;
```

MasterSocket :
retourne le numéro de la socket d'écoute
exemple:

```
msock:=Sockets1.MasterSocket;
```

Text :
permet l'envoi et la réception d'un texte via la socket
exemple :

```
buffer:=Sockets1.Text;                (réception)
Sockets1.Text:='ceci est un message';  (émission)
```

NonBlocking :
initialisé à FALSE pour le mode bloquant et TRUE pour le mode non-bloquant

Timeout :
quand la socket et bloquée, cette valeur spécifie le temps pendant lequel la socket est bloquée. Par défaut, la valeur est 30 (secondes). La valeur zéro crée une attente infinie.

OOB :
au départ, envoie le texte en urgent
à la réception, reçoit des données urgentes
exemple :

```
buffer := Sockets1.OOB;
sockets1.OOB:='ABORT'; {envoi le terme ABORT en urgence}
```

Les méthodes

Sommaire des méthodes implémentées :

SConnect	Connect to listening server
SListen	Listen on Port
SCancelListen	Cancel listen request
SAccept	Accept client connection
SClose	Close sockets
SReceive	Receive PChar data
SSend	Send PChar data
GetPort	Get local port of SocketNumber
GetIPAddr	Get local IP Address
GetPeerPort	Get partner's port assignment
GetPeerIPAddr	Get partner's IP Address

SConnect :
se connecte sur le système distant spécifié par les propriétés IPAddr et Port.
exemple : Sockets1.SConnect;

SListen :
se met en écoute sur le port spécifié par la propriété Port.
exemple : Sockets1.SListen;

SCancelListen :
arrête l'écoute
exemple : Sockets1.SCancelListen;

SAccept :
accepte la requête d'un client. Se trouve en principe dans l'événement OnSessionAvailable.
exemple : soc:=Sockets1.SAccept;

SClose :
ferme la socket
exemple : Sockets1.Sclose;

SReceive :
reçoit des données du correspondant
exemple : len:=Sockets1.SReceive(Sockets1.SocketNumber, buffer,size_buffer);

SSend :
envoie des données au correspondant
exemple : len:=Sockets1.SSEnd(Sockets1.SocketNumber, buffer, size_buffer);

GetPort :
retourne le numéro du port utilisé par la socket spécifié en argument.

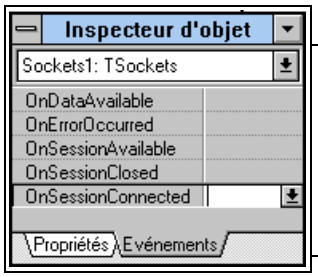
GetIPAddr :
retourne l'adresse IP de la socket spécifiée en argument

GetPeerPort :
retourne le port utilisé par le correspondant sur cette socket

GetPeerIPAddr :
retourne l'adresse IP du correspondant utilisant la socket spécifiée en argument.

Les événements

Sommaire des événements implémentés :



OnDataAvailable	Called when data is available to be received on the socket
OnSessionAvailable	Called when a session is available to be accepted
OnSessionClosed	Called when a connection is lost
OnSessionConnected	Called when an SConnect completes
OnErrorOccurred	Called on error conditions

OnDataAvailable :
envoyé quand une donnée est prête pour être reçu par le correspondant

on utilise :
buffer:=Sockets1.Text;
ou une méthode SReceive pour récupérer les données.

OnSessionAvailable :
envoyé quand un client fait une demande de connexion.
On peut y trouver la méthode SAccept.

OnSessionClosed :
envoyé quand le correspondant a fermé une socket sur vous.
Normalement, on ferme la socket de notre coté dans ce cas.

OnSessionConnected :
envoyé quand la méthode SConnect est OK.
On y trouve généralement les premiers échanges de la conversation.

OnErrorOccurred :
envoyé en cas de problème sur la socket.
Si cette procédure n'est pas définie, DELPHI affiche le message système et le programme est arrêté.

UTILISATION DE L'OBJET SOCKET

Pour faire le parallèle avec UNIX, les deux schémas suivants reprennent les étapes nécessaires à la création et à l'utilisation des sockets en mode connecté. Deux exemples montrent un cas concret de développement.

Client :

SConnect;

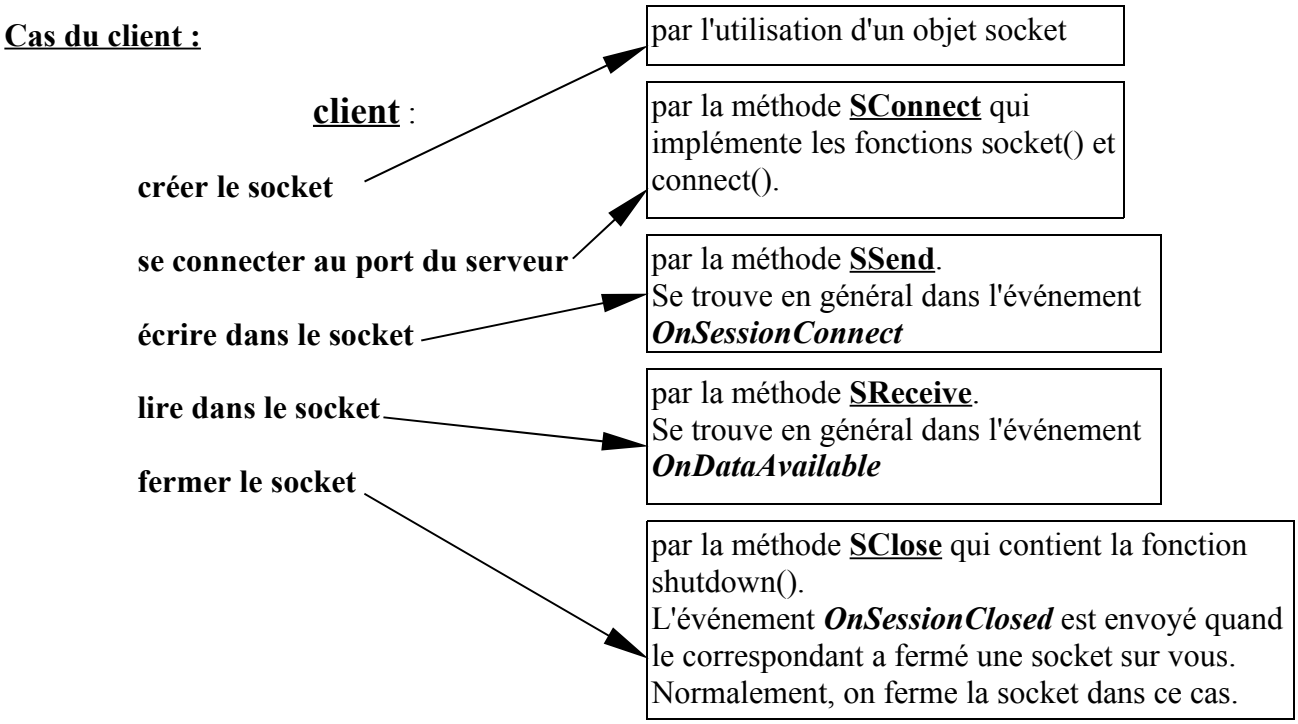
OnSessionConnect	SSend;
OnDataAvailable	SReceive; .... {traitement} .... SSend;
OnErrorOccured	SClose;

Serveur :

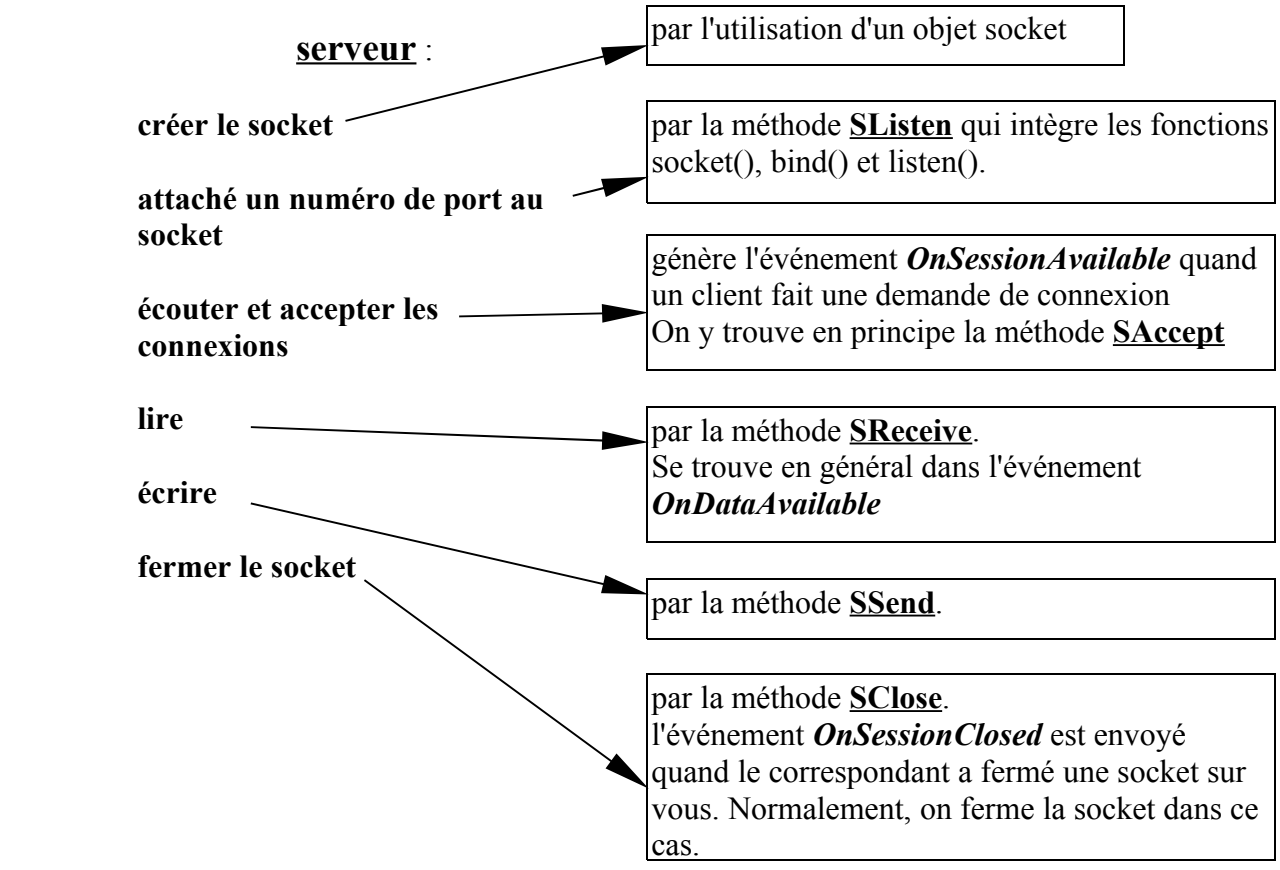
SListen;

OnSessionAvailable	SAccept;
OnDataAvailable	Text; SReceive; ..... {traitement} .... SSend;
OnSessionClose	SClose;
OnErrorOccured	SClose; SCancelListen;

Cas du client :



Cas du serveur :



Enfin, l'événement ***OnErrorOccured*** est envoyé en cas de problème sur le socket.

ECRITURE D'UN CLIENT ET D'UN SERVEUR

Exemple simple :

Pour le client :

```
.....
implementation
{$R *.DFM}

{click sur le bouton CONNEXION}
procedure TForm1.ConnexionClick(Sender: TObject);
begin
        Socket1.SConnect;
end;

procedure TForm1.Socket1Connect(Sender: TObject; Socket: Word);
var
        buffer:pchar;
        tampon:array[0..20] of char;
        taille:integer;
begin
        taille:=20;
        buffer:=@tampon;
        strpcopy(buffer, 'essai de connexion');
        Connexion.Caption:='Connecté';
        Socket1.SSend(Socket1.SocketNumber,buffer, taille);
end;

procedure TForm1.Socket1DataAvailable(Sender: TObject; Socket: Word);
var
        buffer:array[0..20] of char;
        taille:integer;
begin
        taille:=sizeof(buffer);
        Socket1.SReceive(Socket1.SocketNumber, buffer, taille);
        Edit1.Text:=StrPas(buffer);
end;

procedure TForm1.Socket1Close(Sender: TObject; Socket: Word);
begin
        Socket1.SClose;
end;

procedure TForm1.Socket1ErrorOccurred(Sender: TObject; Error: Integer;
        Msg: String);
begin
        Edit1.Text:=Msg;
end;

end.
```

Pour le serveur :

```
        .....
implementation
{$R *.DFM}

procedure TForm1.ConnectClick(Sender: TObject);
begin
        sockets1.Slisten;
end;

procedure TForm1.Sockets1SessionAvailable(Sender: TObject; Socket: Word);
var
        soc:TSocket;
begin
        Memo1.Lines.Add('Demande de socket en n° '+inttostr(socket));
        soc:=Sockets1.SAccept;
        Memo1.Lines.Add('Socket acceptée en n° '+inttostr(soc));
end;

procedure TForm1.Sockets1DataAvailable(Sender: TObject; Socket: Word);
var
        t:PChar;
        l:integer;
begin
        t:=MemAlloc(4096);
        l:=4096;
        l:=sockets1.sreceive(socket,t,l);
        t[l]:=chr(0);
        Memo1.Lines.Add('Message reçu:'+strpas(t)+' en socket n°'+inttostr(socket));
        Sockets1.SSend(socket, t, l);
        Dispose(t); {libère l'espace alloué à t}
end;

procedure TForm1.Sockets1SessionClosed(Sender: TObject; Socket: Word);
begin
        Memo1.Lines.add('Client a fermé la socket n°'+inttostr(Socket));
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
        Sockets1.SClose;
end;

procedure TForm1.Sockets1ErrorOccurred(Sender: TObject; Error: Integer;
        Msg: String);
begin
        Edit2.Text:=Msg;
End ;
End.
```

UTILISATION DES COMPOSANTS INTERNET

Depuis quelques années, le mot "Internet" est sur toutes les lèvres. La puissance de l'Internet est due en partie aux protocoles (langages utilisés par les applications de réseau pour communiquer) permettant à n'importe quel utilisateur d'accéder à des informations et à des applications situées n'importe où dans le monde. Ces mêmes concepts permettent à des sociétés de diffuser des informations internes à tous leurs employés en utilisant des intranets. Un intranet est un réseau privé dont les ressources sont similaires à celles de l'Internet, mais qui ne sont accessibles qu'à des personnes dont l'accès est autorisé.

Nombreux sont ceux pour qui "Internet" et "Web" sont une seule et même chose. Ce n'est pas le cas. Le World Wide Web (WWW ou Web) n'est qu'une des applications ou ensemble de protocoles, utilisant l'Internet comme mécanisme de livraison. L'Internet est le réseau physique et logique qui interconnecte toutes les machines qui y sont reliées. Le protocole de réseau par le biais duquel communiquent les machines de l'Internet est TCP/IP (*Transmission Control Protocol/Internet Protocol*). Les serveurs et browsers Web communiquent au moyen de protocoles de niveau supérieur — principalement HTTP et FTP — pour transmettre les informations entre le client et le serveur.

Il est maintenant possible d'utiliser n'importe quel protocole s'appuyant sur TCP/IP afin d'exploiter l'Internet pour communiquer. De même, le protocole HTPP peut-être utilisé pour permettre à une application de communiquer avec n'importe quelle autre et non uniquement vers le Web. Vous pouvez ainsi développer un jeu de course automobile dans lequel deux conducteurs s'affrontent, les données étant partagées entre les deux à l'aide de HTTP.

Dans ce chapitre, vous découvrirez les concepts de base du Web et la façon de produire, avec Delphi, des applications Web client et serveur robustes. Delphi est un outil très puissant pour ce type de création. Comme ce support ne fera qu'effleurer les possibilités de Delphi en la matière, n'hésitez pas à vous reporter à l'aide en ligne et aux manuels pour connaître les vastes possibilités offertes par Delphi sur le Web.

HTTP

Les deux composantes les plus importantes du Web sont HTTP et HTML (*HyperText Markup Language* ou Langage de marquage hypertexte). HTML n'est pas un langage à proprement parler, mais un standard décrivant le format du contenu. Cela signifie que si vous déclarez qu'un document est conforme au standard HTML, des lecteurs peuvent interpréter certaines balises et leur donner un sens. Prenons par exemple un "document" dont le contenu est le suivant :

```
<HTML>
<B> Ceci est en gras </B> <BR>Ceci non.
</HTML>
```

"Ceci non." est dans la police standard. La balise <B> active la mise en gras et </B> la désactive. La balise
 indique un saut de ligne. Nous n'entrerons pas ici dans le détail de HTML. Il existe de nombreux ouvrages consacrés à sa syntaxe ainsi que des programmes permettant de générer du code HTML.

HTTP est un protocole de réseau client/serveur très puissant. Cette puissance vient en partie du fait qu'il permet à un client et à un serveur de communiquer sans qu'il soit nécessaire de maintenir une connexion de réseau persistante. Une URL (http://www.borland.com par exemple) est un emplacement universel de ressource qui représente un objet présent sur l'Internet. Si vous utilisez un browser Web pour vous rendre à cette URL, la page Web est transférée vers le browser et l'utilisateur peut alors la visualiser. Cependant, une fois que le chargement de la page s'est achevé, la connexion est rompue. On peut dire, en quelque sorte, que le serveur envoie les données par salves successives. Dans le cas d'une application pour laquelle le réseau est inactif la plupart du temps (comme c'est le cas lorsqu'un utilisateur lit une page Web), le principe de salves de données est tout à fait adapté car il permet au serveur de traiter d'autres requêtes d'informations sans qu'il doive maintenir les ressources correspondant à chacune des connexions inactives.

Le protocole HTTP est orienté transaction : le client effectue une demande de données, après quoi le serveur satisfait à sa requête puis achève la connexion. Le contenu demandé peut être de pratiquement n'importe quel type : documents HTML, images, applications et tout autre objet que le client et le serveur "connaissent" tous deux.

Un autre aspect de la puissance de la technologie Web a trait aux URL, qui sont comme les entrées d'un index universel de l'Internet. Celles-ci permettent d'utiliser en conjonction et de manière intégrée les technologies les plus variées. Une application serveur Web écrite en Delphi peut ainsi être intégrée de manière transparente à une application Web écrite en Perl sur une autre machine utilisant un système d'exploitation différent. Ainsi, par exemple, une application serveur Web Delphi pourrait renvoyer des informations concernant un produit à un browser Web. Lorsque l'utilisateur souhaite acheter le produit, il indique son numéro à une application Perl située sur un serveur UNIX, qui est lié au système d'expédition et de distribution de la société de VPC.

Pour utiliser les exemples de serveur présentés dans ce chapitre, vous aurez besoin d'un serveur Web prenant en charge CGI, ISAPI, NSAPI ou WIN-CGI (ou toute combinaison de ces technologies). Le Microsoft Internet Information Server (IIS) ou le Personal Web Server (fourni avec Windows NT 4.0) gèrent ISAPI et CGI. Il existe d'autres serveurs tournant sur Windows 95, tels que celui fourni avec Front Page 97. Nous examinerons par la suite leurs différences.

Pour utiliser des formulaires actifs, vous aurez besoin d'un browser prenant en charge ActiveX (MSIE 3.0 par exemple).

LE CONTENU STATIQUE DE L'INTERNET

A ses débuts, le Web ne contenait pratiquement que des pages statiques. Autrement dit, lorsqu'un browser Web sélectionnait une URL, le serveur Web renvoyait le document HTML correspondant à cette URL. Le code HTML pouvait également contenir des hyperliens vers d'autres pages Web. Pour l'administrateur système, il suffisait d'enregistrer les fichiers HTML dans une structure de fichier hiérarchisée logiquement. Ce paradigme était parfait pour fournir des informations statiques, mais il ne permettait pas l'interactivité.

Une page Web statique simple

```
<HTML>
<TITLE> La cabane du jardinier </TITLE>
<BODY>
<H1> La cabane du jardinier </H1>
<HR>
Avec nous, votre jardin
Les sp&eacute;cialistes mondiaux du rateau
<BR>
<B> Appelez 01-44444444 </B> pour plus d'informations !
<HR>
<A HREF="http://www.jardino.com/tarifs">
Cliquez ici pour voir nos tarifs </A>
</BODY>
</HTML>
```

Lorsque le client (ou le browser) demande http://www.jardino.com/default.htm, son serveur Web renvoie le contenu du fichier default.htm accompagné d'informations de mise à jour. Ceci peut convenir à La cabane du jardinier, mais le résultat n'est pas spectaculaire et le contenu restera le même à chaque visite.

CRÉER DU CONTENU DYNAMIQUE AVEC DELPHI

Imaginons maintenant que l'on souhaite afficher un slogan différent chaque fois que la page est appelée. Un point crucial à garder à l'esprit pour le code HTML généré dynamiquement est que *tout* le traitement s'effectue sur le serveur. Ce dernier ne se contente plus de renvoyer le contenu d'un fichier (comportement statique). Si une requête dynamique est effectuée, il doit traiter un code particulier afin de déterminer ce qui sera envoyé au client. Le serveur peut le traiter au moyen de différentes techniques. Les deux plus courantes consistent à lancer un exécutable indiquant au serveur ce qu'il convient de renvoyer, ou à appeler une DLL qui exécute un code spécifique et en informe le serveur.

Delphi prend en charge quatre types de processus côté serveur pour la création de HTML dynamique.

Processus	DLL ou EXE
ISAPI	DLL
NSAPI	DLL
CGI	EXE
WIN-CGI	EXE

Dans ce chapitre, nous mettrons principalement l'accent sur les ISAPI et les applications CGI. Toutefois, les processus NSAPI sont très similaires aux ISAPI et les programmes WIN-CGI ressemblent aux CGI.

Différences entre ISAPI, NSAPI, CGI et WIN-CGI

Les processus serveur exécutables, tels que les CGI, WIN-CGI, et les DLL en processus, telles que les ISAPI et NSAPI, ont chacun des avantages et des inconvénients spécifiques. Les applications CGI (*Common Gateway Interface*) constituent le premier type d'application produisant du HTML dynamique. Lorsqu'un serveur Web reçoit une requête de traitement d'un CGI, il transmet à l'application CGI toutes les informations provenant du client, au moyen de variables d'environnement et de l'entrée standard (stdin). L'application CGI renvoie le code HTML au client par le biais de la sortie standard (stdout). Le processus est en fait plus complexe, puisque des en-têtes et des commandes peuvent être transférés, mais le principe reste identique.

Pour bien voir comment fonctionne une application CGI, nous allons écrire un exécutable de console standard fonctionnant sous forme d'une application CGI simple (ce n'est pas la meilleure manière d'écrire des applications Web en Delphi, mais cet exemple nous permettra de mieux comprendre ce qui se passe). Le code du Listing suivant dit bonjour puis donne l'heure.

Une fois le programme compilé, il suffit de placer l'exécutable dans un répertoire disposant de privilèges d'exécution sur un serveur Web. L'utilisateur peut alors simplement accéder à l'URL pointant vers l'application.

Un exécutable CGI simple utilisant une application de console

```
Program consolecgi;

uses
        SysUtils;

begin
        writeln('Content-Type:text/html');
        writeln;
        writeln('<H1> Bonjour </H1> <HR>');
        writeln('Il est '+TimeToStr(Time));
end.
```

Cette application se contente d'envoyer les données au client via la sortie standard, au moyen de plusieurs déclarations Writeln*. La simplicité du principe peut sembler séduisante. Cependant, avec cette méthode, vous ne pouvez pas tirer parti du cadre de développement Delphi pour les applications serveur. Le cadre de développement (framework) de serveur Web Delphi, dont nous allons parler un peu plus loin, permet d'utiliser une base de code commune aux exécutables et aux processus Web en processus. Il fournit également des routines qui s'acquittent du plus gros du travail lié au développement d'applications CGI et ISAPI/NSAPI.*

L'une des meilleures raisons de préférer CGI à ISAPI (*Internet Server API*) ou NSAPI (*Netscape Server API*) est que la quasi-totalité des serveurs Web fonctionnant sous Windows peuvent alors utiliser le même exécutable compilé. Cependant, si les performances sont le critère principal, CGI n'est pas le plus indiqué. Chaque fois qu'une application client appelle un programme CGI, le serveur Web doit créer un nouveau processus, puis exécuter l'application CGI, renvoyer le résultat au client, et enfin libérer toutes les ressources impliquées. La charge de travail du serveur est importante, et ce particulièrement s'il est très sollicité. Mieux vaut que le serveur Web exécute votre code dans son propre espace de

processus sans avoir à lancer un nouvel exécutable chaque fois qu'une requête dynamique est envoyée. C'est précisément le principe qui sous-tend les applications ISAPI (et NSAPI). ISAPI est l'API de serveur du serveur Microsoft et ISAPI est son équivalent pour le serveur de Netscape.

Une application ISAPI est une DLL à threads protégés qui s'exécute dans l'espace de processus du serveur Web. Lorsqu'une requête HTTP appelle la DLL ISAPI, le serveur Web prend un thread dans son pool de thread. Quand elle le lance, il s'exécute dans la DLL. Un pool de thread est un ensemble de threads gérés par le serveur Web, qui peut grandir ou rétrécir dynamiquement en fonction de la charge du serveur. Une fois que le résultat a été envoyé au client, le thread est remis à disposition dans le pool. Cette gestion des ressources est bien plus rationnelle que la création d'un nouveau processus pour chaque exécutable. L'un des inconvénients d'ISAPI est que ce code à threads protégés est difficile à écrire et à tester. De plus, dans le cas d'une application ISAPI, une fois la DLL chargée par le serveur, il est nécessaire d'arrêter ce dernier si vous souhaitez remplacer la DLL.

FTP

Le protocole FTP (File Transfer Protocol) permet le transfert de données et de fichiers entre une machine locale et une machine distante.

Voir l'annexe sur les composants INTERNET pour avoir plus d'informations sur le composant CLIENT FTP fourni avec DELPHI 3.

SMTP

Le protocole SMTP sert à développer des applications pour envoyer du courrier. Il implémente le protocole précisé dans la spécification RFC 821, Simple Mail Transfer Protocol. Il offre aux applications un accès à des serveurs de messagerie SMTP et procure des fonctionnalités d'envoi de courrier.

Voir l'annexe sur les composants INTERNET pour avoir plus d'informations sur le composant CLIENT SMTP fourni avec DELPHI 3.

POP3

Le protocole POP3 procure un accès aux serveurs de messagerie Internet en utilisant les spécifications de la RFC 1081, Post Office Protocol. Il peut être utilisé par des développeurs de messagerie Internet ou par des intégrateurs système.

Voir l'annexe sur les composants INTERNET pour avoir plus d'informations sur le composant CLIENT POP3 fourni avec DELPHI 3.

ANNEXE AU CHAPITRE SUR LES COMMUNICATIONS

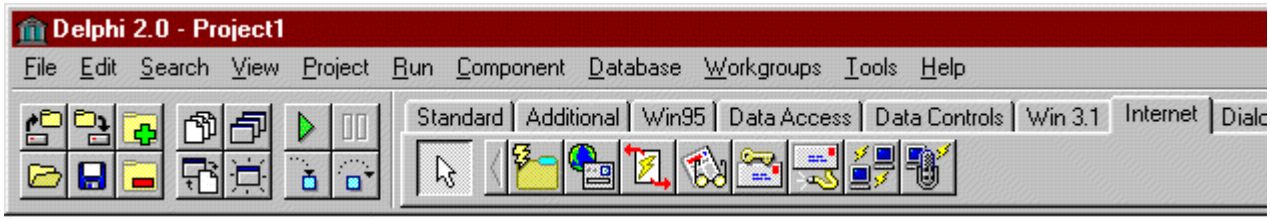
Ou

Comment utiliser les composants INTERNET fournis avec DELPHI 3

DELPHI 3 fournit, dans l'onglet INTERNET, 8 outils de communications qui s'appuient sur TCP/IP. Ces différents outils disposent d'une aide succinte. Lors de la conférence des développeurs BORLAND (1996), les informations suivantes ont étés fournies aux développeurs. Elles vous sont fournies en version originale. Vous pourrez également vous référer à l'aide en ligne de DELPHI pour obtenir d'autres informations (en français).

Using Internet Solutions Pack in Delphi

Borland Developers Conference - 96



Delphi Internet Solutions Pack

With Delphi's new Internet Solutions Pack, Delphi developers can now easily leverage their existing Delphi knowledge and begin to build Internet/Intranet enabled applications. These eight robust ActiveX controls now currently ship with the 2.01 version of Delphi Developer and Delphi Client/Server.

The new Internet Solutions Pack will allow you to develop various Internet/Intranet applications-- without having to first learn and understand WinSock API programming or transfer protocols. These controls wrap all the calls neccessary to perform the service.
This paper describes the functionality of the eight controls and gives reference to the control's properties, methods, and events. It also includes the source code required to wrap the activeX controls and some example code that uses them.

FTP : File Transfer Protocol	HTML : Hyper Text Markup Language
HTTP :HyperText Transport Protocol	NNTP : Networking News Transfer Protocol
POP : Post Office Protocol	SMTP : Simple Mail Transport Protocol
TCP : Transmission Control Protocol	UDP : User Datagram Protocol

FTP - File Transfer Protocol

The FTP (File Transfer Protocol) Client control allows easy file and data transfer between a remote and local machine. FTP is one of the oldest and most used protocols in the Internet. FTP originated to promote sharing of files, access of remote computers, a standard interface independent of the host system, and a reliable data transfer.

There are thousands of FTP archive sites filled with resources including ASCII (text) and binary files over various platforms. The FTP protocol enables the user to log into servers connected over the Internet for the upload or download of files.

The FTP Client Control provides easy access to Internet FTP services by Delphi programmers. Using the FTP control allows you to write standard FTP functionality without the requirement of understanding the details of FTP or the low level WinSock APIs. By placing the control on the form, setting properties, and calling methods you can easily inplement the FTP protocol.

For futher information regarding the FTP implementation, see RFC 959.

Properties:

AppendToFile	This property applies to PutFile and SendDoc to indicate whether the data should be appended to the file (True) or whether the file should be replaced (False).
Busy	Indicates a command is in progress.
DocInput	Object describing input information for the document being transferred.
DocOutput	Object describing output information for the document being transferred.
EnableTimer	Boolean property to enable timer for the specified event.Value is specified in the TimeOut property.
Errors	A collection of errors that can be accessed for details about the last error that occurred. This collection should be used within an Error event if information passed through the Error event is not sufficient.
ListItemNotify	Causes the container to receive events for every directory element received during a List or NameList command. If this property is TRUE, the directory listing is parsed and events activated for every directory element. If this property is FALSE, the list data is sent in blocks to the data target during ProcessData notifications.
NotificationMode	Determines when notification is issued for incoming data. Notification can also be suspended. 0 = COMPLETE: notification is provided when there is a complete response.

	1 = CONTINUOUS: an event is repeatedly activated when new data arrives from the connection.
Operation	Allows you to determine the last method performed that caused data to be received. This property is normally used when processing the DocOutput event. FTPOperationConstants include: ftpFile = 0 ftpList = 1 ftpNameList = 2. Using this property allows you to determine which method activated the DocOutput event, making it possible to distinguish between the various types of data.
Password	Password of current user on the FTP Server.
ProtocolState	This property specifies the current state of the protocol. Constants defined for enum types of ProtocolState property are: ftpBase (default)= 0 (Default) - the state before connection server is established. ftpAuthorization = 1 - authorization is performed. ftpTransaction = 2 - Authorization successful. The client has successfully identified itself to the FTP server.
ProtocolStateString	String representation of ProtocolState.
RemoteDir	The remote directory name. This read only property is set each time a changeDir or a PrintDir method is invoked. This run-time read-only property is not used to set the remote working directory. It provides the convenience of a parsed string containing the current working directory of the remote machine.
RemoteFile	The remote file name used during GetFile and PutFile operations.
RemoteHost	The remote machine to connect to if the remoteHost parameter in the Connect method is missing. You can either provide a host name or an IP address string in dotted format. For example, 127.0.0.1.
RemotePort	The remote port number to which to connect.
ReplyCode	The value of the reply code is a protocol specific number that determines the result of the last request, as returned in the ReplyString property. See RFC 959 for Valid reply codes.
ReplyString	Lists the last reply string sent by the FTP Server to the client as a result of a request. This string contains both a number code and a status string that the server creates for the last command.

State	This property specifies the connection state of the control. prcConnecting = 1 - Connecting. Connect has been requested, waiting for connect acknowledge. prcResolvingHost = 2 - Resolving Host. Occurs when RemoteHost is in name format rather than dot-delimited IP format. prcHostResolved = 3 - Resolved the host. Occurs only if ResolvingHost state has been entered previously. prcConnected = 4 - Connection established. prcDisconnecting = 5 - Connection closed. Disconnect has been initiated. prcDisconnected = 6 - Initial state when protocol object is instantiated, before Connect has been initiated, after a Connect attempt failed or after Disconnect performed.
StateString	A string representation of State.
TimeOut	Timeout value for the specified event.
URL	URL (Universal Resource Locator) string identifying the current document being transferred. The URL format when using the FTP Control is: ***FTP://username.password@host/documentnameand path***
UserId	User identification name for the client on the server.

Methods:

Abort	Requests a FTP Server to abort the last data transfer request. Similar to the FTP RFC-959 ABORT command. This event usually terminates any data connection while leaving the control connection intact.
Accout	Sends account information to remote host. Similar to the FTP RFC-959 ACCT command. Use the ReplyString property to determine the result of this call.
Authenticate	Authenticates the user based on the parameters passed. If no parameters are passed, the UserId and Password properties are used. If neither the UserId or Password is entered, the control uses the URL. When authentication process is terminated, the Authenticate event is activated.
Cancel	Cancels a pending request.
ChangeDir	Requests FTP Server to change the remote host current directory to the specified directory. Similar to the FTP RFC-959 CWD command. Use the ReplyString property to determine the result of this call.

Connect	Initiates a Connect request. The control calls the OnStateChanged event if a connection is established.
CreateDir	Creates the specified directory on the remote host. Similar to the FTP RFC-959 MKD command. Use the ReplyString property to determine the result of this call.
DeleteDir	Deletes the specified directory file from the remote host. Similar to the FTP RFC-959 RMD command. Use the ReplyString property to determine the result of this call.
DeleteFile	Deletes the specified file from the remote host. Similar to the FTP RFC-959 DELE command. Use the ReplyString property to determine the result of this call.
Disconnect	Disconnects session with remote host and terminates any data connection.
GetDoc	A DocOutput related method that requests retrieval of a document identified by a URL. The GetDoc method in FTP means retrieving a file from the server.
GetFile	Gets the specified file from the remote host and places it in the current directory.
Help	Gets FTP help from the remote host. Similar to the FTP RFC-959 HELP command. Use the ReplyString property to determine the result of this call.
List	Requests a detailed directory listing of the specified directory from the remote host. Similar to the FTP RFC-959 LST command. The data from this method is sent to the DocStream interface via the OnDocOutput event. During processing of the OnDocOutput event, the Operation property is set to ftpList. If the ListItemNotify property is set to True, the ListItem event is also generated for every item in the directory listing.
Mode	Sets data transfer mode of remote host. Similar to the FTP RFC-959 MODE command. The FTPModeConstants may have one of the following values. ftpStream = 0 ftpBlock = 1 ftpCompressed = 2
NameList	Requests a directory listing of the specified directory from the remote host. Similar to the FTP RFC 959 NLST command. The data from this method is sent to the DocStream interface via the OnDocOutput event. During processing of the OnDocOutput event, the Operation property is set to ftpList. If the ListItemNotify property is set to True, the ListItem event is also generated for every item in the directory listing.

NOOP	Issues the NOOP command to the server. Use the ReplyString property to determine the result of this call.
ParentDir	Requests the FTP Server change to the parent of the current directory, if one exists. Use the ReplyString property to determine the result of this call.
PrintDir	Requests the FTP Server query the current directory of the remote host. Similar to the FTP RFC-959 PWD command. Use the ReplyString property to determine the result of this call. You will need to parse the ReplyString to determine the directory name. You can also obtain this information from the RemoteDir property.
PutFile	Puts specified file on the Server's current directory.
ReInitialize	Issues the ReInit command to the server. Use the ReplyString property to determine the result of this call.
SendDoc	A DocInput related method that requests sending a document identified by a URL. The SendDoc method in FTP means putting a file on the server.
Site	Issues a Site command to the remote server. This command is used during logon to determine the file system supported on the server. Use the ReplyString property to determine the result of this call.
State	Requests status from the remote host. Similar to the FTP RFC-959 STAT command. Use the ReplyString property to determine the result of this call during the State event notification.
System	Issues a system command to the remote server. It is similar to the FTP RFC-959 SYST command.
Type	Issues a Type command to the remote server. This command is entered prior to a data transfer to set the transfer type. The server attempts to use this value for data representation if it is supported. Possible values are: ftpAscii = 0 ftpImage = 2 ftpEBCDIC = 1 ftpBinary = 3

Events:

OnAbort	This event is activated after the Abort method is called. It aborts any active FTP process.
OnAccout	This event is activated after the Account method is called. It requests the remote host set the account to the one specified.

OnBusy	This event is activated when a command is in progress or when a command has completed.
OnCancel	This event is activated after a cancellation request has been completed and satisfied. After this event the object's state changes to idle.
OnChangeDir	This event is activated after the ChangeDir method is called. It changes the current working directory.
OnCreateDir	This event is activated after the CreateDir method is called. It creates a new directory.
OnDeleteDir	This event is activated after the DeleteDir method is called. It deletes the specified directory on the remote host.
OnDelFile	This event is activated after the DeleteFile method is called. It deletes the specified file located in the path specified on the remote host.
OnDocInput	A DocInput related event that indicates the input data has been transferred. The DocInput event can be used in its basic form for notification during transfer.
OnDocOutput	A DocOutput related event indicating that output data has been transferred. The DocOutput event can be used in its basic form for notification during transfer.
OnError	This event is activated when an error occurs in background processing.

FTP Error Codes : The following error codes apply only to the FTP ActiveX Control.

Error Code	Error Message
2104	Port Command Failed. Unable to open Port.
2105	Abort Command Failed. Unable to Abort last command.
2106	Account Command Failed. Unable to complete.
2107	Change Directory Command Failed. Unable to change to specified directory.
2108	Connect Command Failed. Unable to connect to remote host.
2109	Create Directory Command Failed. Unable to create specified directory.
2110	Delete Directory Command Failed. Unable to delete specified directory.
2111	Delete File Command Failed. Unable to delete specified file.
2112	Disconnect Command Failed. Unable to disconnect from remote host.

	2113	Get File Command Failed. Unable to retrieve specified file.
	2114	Help Command Failed. Unable to retrieve help from remote host.
	2115	NOOP Command Failed. Control connection error.
	2116	Name List Command Failed. Unable to retrieve Named list; possible data connection error.
	2117	List Command Failed. Unable to retrieve detailed list; possible data connection error.
	2118	Parent directory Command Failed. Unable to change directory up.
	2119	Print Directory Command Failed. Unable to print current directory of remote host.
	2120	Put File Command Failed. Unable to put file on remote host.
	2121	Put Unique File Command Failed. Unable to put unique file on remote host.
	2122	Reinitialize Command Failed. Unable to reinitialize login on remote host.
	2123	Rename File Command Failed. Unable to rename specified file on remote host.
	2124	Retrieve File Command Failed. Unable to retrieve specified file from remote host.
	2125	Status Command Failed. Unable to retrieve status from remote host.
	2126	System Command Failed. Unable to issue SYST command to remote host.
	2127	Type Command Failed. Unable to set transfer type on remote host.
	2128	Error setting OutputDocStream property
	2129	Error setting InputDocStream property
	2157	Command not implemented.
	2171	Maximum connection() reached.

OnExecute	This event is activated after the Execute method is called. It issues a command to the server for processing.
OnHelp	This event is activated after the HELP method is called. It requests the remote host send help information with the specified HELP parameters.
OnListItem	This event is activated for every element in a directory listing when the ListItemNotify property is set to TRUE. This lets you parse the directory elements after issuing a List or NameList command.
OnLog	This event is activated when the Logging property is set to TRUE.

OnMode	This event is activated after the Mode method is called. It sets the remote host data transfer mode to the mode specified.
OnNoop	This event is activated after the NOOP method is called. It requests an OK reply from the server.
OnParentDir	This event is activated after the ParentDir method is called. It changes the current directory on the remote host to the parent directory, if one exists.
OnPrintDir	This event is activated after the PrintDir method is called. It requests the remote host print the working directory.
OnProtocolStateChanged	This event is activated whenever the protocol state changes.
OnReinitialize	This event is activated whenever the ReInit method is called.
OnSite	This event is activated after the Site method is called. It requests directory and file formatting information from the remote host.
OnStateChanged	This event is activated whenever the state of the transport state changes.
OnStatus	This event is activated whenever the status of the transport state changes.
OnSystem	This event is activated after the System method is called. It requests the type of operating system on the server.
OnTimeout	This event is activated when the timer for the specified event expires. See Timeout property for pre-defined events.
OnType	This event is activated the Type method is called. Specifies how the remote host should handle the transferred data.

As Declared in ISP.PAS

```
{ TFTP }

TFTPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
TFTPTimeout = procedure(Sender: TObject; event: Smallint; var Continue:
TOleBool) of object;
TFTPStateChanged = procedure(Sender: TObject; State: Smallint) of object;
TFTPProtocolStateChanged = procedure(Sender: TObject; ProtocolState:
Smallint) of object;
TFTPBusy = procedure(Sender: TObject; isBusy: TOleBool) of object;
TFTPDocInput = procedure(Sender: TObject; const DocInput: Variant) of
object;
TFTPDocOutput = procedure(Sender: TObject; const DocOutput: Variant) of
object;
TFTPListItem = procedure(Sender: TObject; const Item: Variant) of object;
```

```
TFTP = class(TOleControl)
  private
    FOnError: TFTPError;
    FOnTimeout: TFTPTimeout;
    FOnCancel: TNotifyEvent;
    FOnStateChanged: TFTPStateChanged;
    FOnProtocolStateChanged: TFTPProtocolStateChanged;
    FOnBusy: TFTPBusy;
    FOnLog: TNotifyEvent;
    FOnDocInput: TFTPDocInput;
    FOnDocOutput: TFTPDocOutput;
    FOnAbort: TNotifyEvent;
    FOnAccount: TNotifyEvent;
    FOnChangeDir: TNotifyEvent;
    FOnCreateDir: TNotifyEvent;
    FOnDeleteDir: TNotifyEvent;
    FOnDelFile: TNotifyEvent;
    FOnHelp: TNotifyEvent;
    FOnmode: TNotifyEvent;
    FOnNoop: TNotifyEvent;
    FOnParentDir: TNotifyEvent;
    FOnPrintDir: TNotifyEvent;
    FOnExecute: TNotifyEvent;
    FOnStatus: TNotifyEvent;
    FOnReinitialize: TNotifyEvent;
    FOnSystem: TNotifyEvent;
    FOnSite: TNotifyEvent;
    FOnType: TNotifyEvent;
    FOnListItem: TFTPListItem;
    function Get_Timeout(event: Smallint): Integer; stdcall;
    procedure Set_Timeout(event: Smallint; Value: Integer); stdcall;
    procedure Set_EnableTimer(event: Smallint; Value: TOleBool); stdcall;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure Cancel; stdcall;
    procedure Connect(const RemoteHost, RemotePort: Variant); stdcall;
    procedure Authenticate(const UserId, Password: Variant); stdcall;
    procedure SendDoc(const URL, Headers, InputData, InputFile, OutputFile:
        Variant); stdcall;
    procedure GetDoc(const URL, Headers, OutputFile: Variant); stdcall;
    procedure Abort; stdcall;
    procedure Account(const Account: string); stdcall;
    procedure ChangeDir(const directory: string); stdcall;
    procedure CreateDir(const directory: string); stdcall;
    procedure DeleteDir(const directory: string); stdcall;
    procedure DeleteFile(const FileName: string); stdcall;
    procedure Quit; stdcall;
    procedure Help(const Help: string); stdcall;
    procedure Noop; stdcall;
    procedure mode(ftpMode: TOleEnum); stdcall;
    procedure Type_(ftpType: TOleEnum); stdcall;
    procedure List(const List: string); stdcall;
    procedure NameList(const NameList: string); stdcall;
    procedure ParentDir; stdcall;
    procedure PrintDir; stdcall;
    procedure Execute(const Execute: string); stdcall;
    procedure Status(const Status: string); stdcall;
    procedure PutFile(const srcFileName, destFileName: string); stdcall;
```

```
  procedure Reinitialize; stdcall;
  procedure System; stdcall;
  procedure GetFile(const srcFileName, destFileName: string); stdcall;
  procedure PutFileUnique(const FileName: string); stdcall;
  procedure Site(const Site: string); stdcall;
  property State: Smallint index 503 read GetSmallintProp;
  property ProtocolState: Smallint index 504 read GetSmallintProp;
  property ReplyString: string index 505 read GetStringProp;
  property ReplyCode: Integer index 506 read GetIntegerProp;
  property Errors: Variant index 508 read GetVariantProp;
  property Busy: TOleBool index 509 read GetOleBoolProp;
  property StateString: string index 511 read GetStringProp;
  property ProtocolStateString: string index 512 read GetStringProp;
  property DocInput: Variant index 1002 read GetVariantProp;
  property DocOutput: Variant index 1003 read GetVariantProp;
  property Operation: TOleEnum index 5 read GetOleEnumProp;
  property Timeout[event: Smallint]: Integer read Get_Timeout write
      Set_Timeout;
  property EnableTimer[event: Smallint]: TOleBool write Set_EnableTimer;
published
  property RemoteHost: string index 0 read GetStringProp write
      SetStringProp stored False;
  property RemotePort: Integer index 502 read GetIntegerProp write
      SetIntegerProp stored False;
  property NotificationMode: Smallint index 510 read GetSmallintProp
      write SetSmallintProp stored False;
  property Logging: TOleBool index 514 read GetOleBoolProp write
      SetOleBoolProp stored False;
  property UserId: string index 601 read GetStringProp write
      SetStringProp stored False;
  property Password: string index 602 read GetStringProp write
      SetStringProp stored False;
  property URL: string index 1001 read GetStringProp write SetStringProp
      stored False;
  property AppendToFile: TOleBool index 1 read GetOleBoolProp write
      SetOleBoolProp stored False;
  property ListItemNotify: TOleBool index 2 read GetOleBoolProp write
      SetOleBoolProp stored False;
  property RemoteFile: string index 3 read GetStringProp write
      SetStringProp stored False;
  property OnError: TFTPError read FOnError write FOnError;
  property OnTimeout: TFTPTimeout read FOnTimeout write FOnTimeout;
  property OnCancel: TNotifyEvent read FOnCancel write FOnCancel;
  property OnStateChanged: TFTPStateChanged read FOnStateChanged write
      FOnStateChanged;
  property OnProtocolStateChanged: TFTPProtocolStateChanged read
      FOnProtocolStateChanged write FOnProtocolStateChanged;
  property OnBusy: TFTPBusy read FOnBusy write FOnBusy;
  property OnLog: TNotifyEvent read FOnLog write FOnLog;
  property OnDocInput: TFTPDocInput read FOnDocInput write FOnDocInput;
  property OnDocOutput: TFTPDocOutput read FOnDocOutput write
      FOnDocOutput;
  property OnAbort: TNotifyEvent read FOnAbort write FOnAbort;
  property OnAccount: TNotifyEvent read FOnAccount write FOnAccount;
  property OnChangeDir: TNotifyEvent read FOnChangeDir write
      FOnChangeDir;
  property OnCreateDir: TNotifyEvent read FOnCreateDir write
      FOnCreateDir;
  property OnDeleteDir: TNotifyEvent read FOnDeleteDir write
      FOnDeleteDir;
  property OnDelFile: TNotifyEvent read FOnDelFile write FOnDelFile;
```

```
    property OnHelp: TNotifyEvent read FOnHelp write FOnHelp;
    property Onmode: TNotifyEvent read FOnmode write FOnmode;
    property OnNoop: TNotifyEvent read FOnNoop write FOnNoop;
    property OnParentDir: TNotifyEvent read FOnParentDir write
        FOnParentDir;
    property OnPrintDir: TNotifyEvent read FOnPrintDir write FOnPrintDir;
    property OnExecute: TNotifyEvent read FOnExecute write FOnExecute;
    property OnStatus: TNotifyEvent read FOnStatus write FOnStatus;
    property OnReinitialize: TNotifyEvent read FOnReinitialize write
        FOnReinitialize;
    property OnSystem: TNotifyEvent read FOnSystem write FOnSystem;
    property OnSite: TNotifyEvent read FOnSite write FOnSite;
    property OnType: TNotifyEvent read FOnType write FOnType;
    property OnListItem: TFTPListItem read FOnListItem write FOnListItem;
  end;
```

Example:

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Buttons, StdCtrls, ComCtrls, OleCtrls, Menus, ExtCtrls, ISP;

const
  FTPServer = 0;
  Folder = 1;
  OpenFolder = 2;

type
  TMainForm = class(TForm)
    Bevel1: TBevel;
    Panel1: TPanel;
    Panel2: TPanel;
    Panel3: TPanel;
    StatusBar: TStatusBar;
    FileList: TListView;
    DirTree: TTreeView;
    ConnectBtn: TSpeedButton;
    FTP: TFTP;
    RefreshBtn: TSpeedButton;
    MainMenu1: TMainMenu;
    FileMenu: TMenuItem;
    FileNewItem: TMenuItem;
    FileDeleteItem: TMenuItem;
    FileRenameItem: TMenuItem;
    N2: TMenuItem;
    FileExitItem: TMenuItem;
    View1: TMenuItem;
    ViewLargeItem: TMenuItem;
    ViewSmallItem: TMenuItem;
    ViewListItem: TMenuItem;
    ViewDetailsItem: TMenuItem;
    N1: TMenuItem;
    ViewRefreshItem: TMenuItem;
    FilePopup: TPopupMenu;
    DeleteItem: TMenuItem;
    RenameItem: TMenuItem;
    CopyItem: TMenuItem;
    Bevel2: TBevel;
```

```
    Label1: TLabel;
    Bevel3: TBevel;
    Bevel5: TBevel;
    Label2: TLabel;
    SaveDialog1: TSaveDialog;
    CopyButton: TSpeedButton;
    LargeBtn: TSpeedButton;
    SmallBtn: TSpeedButton;
    ListBtn: TSpeedButton;
    DetailsBtn: TSpeedButton;
    Tools1: TMenuItem;
    ToolsConnectItem: TMenuItem;
    ToolsDisconnectItem: TMenuItem;
    FileCopyItem: TMenuItem;
    PasteFromItem: TMenuItem;
    OpenDialog1: TOpenDialog;
    SmallImages: TImageList;
    procedure ConnectBtnClick(Sender: TObject);
    procedure FTPListItem(Sender: TObject; const Item: Variant);
    procedure FTPProtocolStateChanged(Sender: TObject;
      ProtocolState: Smallint);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure FTPBusy(Sender: TObject; isBusy: Wordbool);
    procedure DirTreeChange(Sender: TObject; Node: TTreeNode);
    procedure RefreshBtnClick(Sender: TObject);
    procedure DirTreeChanging(Sender: TObject; Node: TTreeNode;
      var AllowChange: Boolean);
    procedure FTPStateChanged(Sender: TObject; State: Smallint);
    procedure Open1Click(Sender: TObject);
    procedure FileExitItemClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure ViewLargeItemClick(Sender: TObject);
    procedure ViewSmallItemClick(Sender: TObject);
    procedure ViewListItemClick(Sender: TObject);
    procedure ViewDetailsItemClick(Sender: TObject);
    procedure ViewRefreshItemClick(Sender: TObject);
    procedure CopyItemClick(Sender: TObject);
    procedure ToolsDisconnectItemClick(Sender: TObject);
    procedure FileNewItemClick(Sender: TObject);
    procedure DeleteItemClick(Sender: TObject);
    procedure PasteFromItemClick(Sender: TObject);
    procedure FilePopupPopup(Sender: TObject);
    procedure FileMenuClick(Sender: TObject);
    procedure FileDeleteItemClick(Sender: TObject);
  private
    Root: TTreeNode;
    function CreateItem(const FileName, Attributes, Size, Date: Variant):
TListItem;
    procedure Disconnect;
  public
    function NodePath(Node: TTreeNode): String;
  end;

var
  MainForm: TMainForm;
  UserName,
  Pwd: String;

implementation
{$R *.DFM}
```

```
uses ShellAPI, UsrInfo;

function FixCase(Path: String): String;
var
  OrdValue: byte;
begin
  if Length(Path) = 0 then exit;
  OrdValue := Ord(Path[1]);
  if (OrdValue >= Ord('a')) and (OrdValue <= Ord('z')) then
    Result := Path
  else
  begin
    Result := LowerCase(Path);
    Result[1] := UpCase(Result[1]);
  end;
end;

procedure TMainForm.ConnectBtnClick(Sender: TObject);
begin
  if FTP.State = prcConnected then Disconnect;
  ConnectForm := TConnectForm.Create(Self);
  try
    if ConnectForm.ShowModal = mrOk then
      with FTP, ConnectForm do
      begin
        UserName := UserNameEdit.Text;
        Pwd := PasswordEdit.Text;
        RemoteHost := RemoteHostEdit.Text;
        RemotePort := StrToInt(RemotePortEdit.Text);
        Connect(RemoteHost, RemotePort);
        Root := DirTree.Items.AddChild(nil, RemoteHost);
        Root.ImageIndex := FTPServer;
        Root.SelectedIndex := FTPServer;
        DirTree.Selected := Root;
      end;
  finally
    ConnectForm.Free;
  end;
end;

procedure TMainForm.FTPListItem(Sender: TObject; const Item: Variant);
var
  AnItem: TListItem;
  Node: TTreeNode;
begin
  CreateItem(Item.FileName, Item.Attributes, Item.Size, Item.Date);
  if Item.Attributes = 1 then
    if DirTree.Selected <> nil then
     begin
       if DirTree.Selected <> nil then
         Node := DirTree.Selected.GetFirstChild
       else
         Node := nil;
       while Node <> nil do
         if CompareText(Node.Text, Item.FileName) = 0 then
           exit
         else
           Node := DirTree.Selected.GetNextChild(Node);
       if Node = nil then
       begin
         Node := DirTree.Items.AddChild(DirTree.Selected, Item.FileName);
```

```
          Node.ImageIndex := Folder;
          Node.SelectedIndex := OpenFolder;
        end;
      end
      else
        DirTree.Items.AddChild(Root, Item.FileName);
end;

procedure TMainForm.FTPProtocolStateChanged(Sender: TObject;
  ProtocolState: Smallint);
begin
  case ProtocolState of
    ftpAuthentication: FTP.Authenticate(UserName, Pwd);
    ftpTransaction: FTP.List('/');
  end;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  FTP.Cancel;
  FTP.Quit;
  while FTP.Busy do Application.ProcessMessages;
end;

function TMainForm.CreateItem(const FileName, Attributes, Size, Date:
Variant): TListItem;
var
  Ext: String;
  ShFileInfo: TSHFILEINFO;
begin
  Result := FileList.Items.Add;
  with Result do
  begin
    Caption := FixCase(Trim(FileName));
    if Size > 0 then
    begin
      if Size div 1024 <> 0 then
      begin
        SubItems.Add(IntToStr(Size div 1024));
        SubItems[0] := SubItems[0] + 'KB';
      end
      else
        SubItems.Add(Size);
    end
    else
      SubItems.Add('');
    if Attributes = '1' then
    begin
      SubItems.Add('File Folder');
      ImageIndex := 3;
    end
    else
    begin
      Ext := ExtractFileExt(FileName);
      ShGetFileInfo(PChar('c:\*' + Ext), 0, SHFileInfo, SizeOf(SHFileInfo),
        SHGFI_SMALLICON or SHGFI_SYSICONINDEX or SHGFI_TYPENAME);
      if Length(SHFileInfo.szTypeName) = 0 then
      begin
        if Length(Ext) > 0 then
        begin
          System.Delete(Ext, 1, 1);
```

```
          SubItems.Add(Ext + ' File');
        end
        else
          SubItems.Add('File');
      end
      else
        SubItems.Add(SHFileInfo.szTypeName);
      ImageIndex := SHFileInfo.iIcon;
    end;
    SubItems.Add(Date);
  end;
end;

procedure TMainForm.Disconnect;
begin
  FTP.Quit;
  Application.ProcessMessages;
end;

procedure TMainForm.FormCreate(Sender: TObject);
var
  SHFileInfo: TSHFileInfo;
  i: Char;
  Node: TTreeNode;
begin
  with DirTree do
  begin
    DirTree.Images := SmallImages;
    SmallImages.ResourceLoad(rtBitmap, 'IMAGES', clOlive);
  end;
  with FileList do
  begin
    SmallImages := TImageList.CreateSize(16,16);
    SmallImages.ShareImages := True;
    SmallImages.Handle := ShGetFileInfo('*.*', 0, SHFileInfo,
      SizeOf(SHFileInfo), SHGFI_SMALLICON or SHGFI_ICON or
      SHGFI_SYSICONINDEX);
    LargeImages := TImageList.Create(nil);
    LargeImages.ShareImages := True;
    LargeImages.Handle := ShGetFileInfo('*.*', 0, SHFileInfo,
      SizeOf(SHFileInfo), SHGFI_LARGEICON or SHGFI_ICON or
      SHGFI_SYSICONINDEX);
  end;
end;

procedure TMainForm.FTPBusy(Sender: TObject; isBusy: Wordbool);
begin
  if isBusy then
  begin
    Screen.Cursor := crHourGlass;
    FileList.Items.BeginUpdate;
    FileList.Items.Clear;
  end
  else
  begin
    Screen.Cursor := crDefault;
    FileList.Items.EndUpdate;
  end;
end;
```

```
function TMainForm.NodePath(Node: TTreeNode): String;
begin
  if Node = Root then
    Result := '.'
  else
    Result := NodePath(Node.Parent) + '/' + Node.Text;
end;

procedure TMainForm.DirTreeChange(Sender: TObject; Node: TTreeNode);
var
  NP: String;
  i: Integer;
begin
  if (FTP.State = prcDisconnected) or FTP.Busy then exit;
  if Node <> nil then
  begin
    NP := NodePath(DirTree.Selected);
    FTP.List(NP);
    Label2.Caption := Format('Contents of: ''%s/''',[NP]);
  end;
end;

procedure TMainForm.RefreshBtnClick(Sender: TObject);
begin
  FTP.List(NodePath(DirTree.Selected));
end;

procedure TMainForm.DirTreeChanging(Sender: TObject; Node: TTreeNode;
  var AllowChange: Boolean);
begin
  AllowChange := not FTP.Busy;
end;

procedure TMainForm.FTPStateChanged(Sender: TObject; State: Smallint);
begin
  with FTP, Statusbar.Panels[0] do
    case State of
      prcConnecting   : Text := 'Connecting';
      prcResolvingHost: Text := 'Connecting';
      prcHostResolved : Text := 'Host resolved';
      prcConnected    :
        begin
          Text := 'Connected to: ' + RemoteHost;
          ConnectBtn.Hint := 'Disconnect';
          FileNewItem.Enabled := True;
          ViewLargeItem.Enabled := True;
          ViewSmallItem.Enabled := True;
          ViewListItem.Enabled := True;
          ViewDetailsItem.Enabled := True;
          ViewRefreshItem.Enabled := True;
          ToolsDisconnectItem.Enabled := True;
          LargeBtn.Enabled := True;
          SmallBtn.Enabled := True;
          ListBtn.Enabled := True;
          DetailsBtn.Enabled := True;
          RefreshBtn.Enabled := True;
        end;
      prcDisconnecting: Text := 'Disconnecting';
      prcDisconnected :
        begin
          Text := 'Disconnected';
```

```
          ConnectBtn.Hint := 'Connect';
          DirTree.Items.Clear;
          FileNewItem.Enabled := False;
          ViewLargeItem.Enabled := False;
          ViewSmallItem.Enabled := False;
          ViewListItem.Enabled := False;
          ViewDetailsItem.Enabled := False;
          ViewRefreshItem.Enabled := False;
          ToolsDisconnectItem.Enabled := False;
          LargeBtn.Enabled := False;
          SmallBtn.Enabled := False;
          ListBtn.Enabled := False;
          DetailsBtn.Enabled := False;
          RefreshBtn.Enabled := False;
        end;
    end;
end;

procedure TMainForm.Open1Click(Sender: TObject);
begin
  FTP.Quit;
  DirTree.Items.BeginUpdate;
  try
    DirTree.Items.Clear;
  finally
    DirTree.Items.EndUpdate;
  end;
end;

procedure TMainForm.FileExitItemClick(Sender: TObject);
begin
  FTP.Quit;
  Application.ProcessMessages;
  while FTP.Busy do Application.ProcessMessages;
  Close;
end;

procedure TMainForm.FormResize(Sender: TObject);
begin
  Statusbar.Panels[0].Width := Width - 150;
end;

procedure TMainForm.ViewLargeItemClick(Sender: TObject);
begin
  FileList.ViewStyle := vsIcon;
end;

procedure TMainForm.ViewSmallItemClick(Sender: TObject);
begin
  FileList.ViewStyle := vsSmallIcon;
end;

procedure TMainForm.ViewListItemClick(Sender: TObject);
begin
  FileList.ViewStyle := vsList;
end;

procedure TMainForm.ViewDetailsItemClick(Sender: TObject);
begin
  FileList.ViewStyle := vsReport;
end;
```

```
procedure TMainForm.ViewRefreshItemClick(Sender: TObject);
begin
  DirTreeChange(nil, DirTree.Selected);
end;

procedure TMainForm.CopyItemClick(Sender: TObject);
begin
  SaveDialog1.FileName := FileList.Selected.Caption;
  if SaveDialog1.Execute then
      FTP.GetFile(NodePath(DirTree.Selected) + '/' +
        FileList.Selected.Caption, SaveDialog1.FileName);
end;

procedure TMainForm.ToolsDisconnectItemClick(Sender: TObject);
begin
  DisConnect;
end;

procedure TMainForm.FileNewItemClick(Sender: TObject);
var
  DirName: String;
begin
  if InputQuery('Input Box', 'Prompt', DirName) then
    FTP.CreateDir(NodePath(DirTree.Selected) + '/' + DirName);
end;

procedure TMainForm.DeleteItemClick(Sender: TObject);
begin
  if ActiveControl = DirTree then
    FTP.DeleteDir(NodePath(DirTree.Selected));
  if ActiveControl = FileList then
    FTP.DeleteFile(NodePath(DirTree.Selected) + '/' +
        FileList.Selected.Caption);
end;

procedure TMainForm.PasteFromItemClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    FTP.PutFile(OpenDialog1.FileName, NodePath(DirTree.Selected));
end;

procedure TMainForm.FilePopupPopup(Sender: TObject);
begin
  CopyItem.Enabled := (ActiveControl = FileList) and (FileList.Selected <>
        nil);
  PasteFromItem.Enabled := (ActiveControl = DirTree) and (DirTree.Selected
        <> nil);
  DeleteItem.Enabled := (ActiveControl = FileList) and (FileList.Selected
        <> nil);
  RenameItem.Enabled := (ActiveControl = FileList) and (FileList.Selected
        <> nil);
end;

procedure TMainForm.FileMenuClick(Sender: TObject);
begin
  FileCopyItem.Enabled := (ActiveControl = FileList) and (FileList.Selected
        <> nil);
  FileDeleteItem.Enabled := (ActiveControl = FileList) and
        (FileList.Selected <> nil);
  FileRenameItem.Enabled := (ActiveControl = FileList) and
        (FileList.Selected <> nil);
```

```
end;

procedure TMainForm.FileDeleteItemClick(Sender: TObject);
begin
  if (DirTree.Selected <> nil) and (FileList.Selected <> nil) then
    FTP.DeleteFile(FileList.Selected.Caption);
end;

end.
```

HTML - Hypertext Markup Language

The HTML Control offers you the ability to build applications incorporating HTML ("Web") browsing capabilities. The control supports HTML 2.x tags and extensions from both the Netscape 2.0 and Microsoft Explorer 2.0 browsers. The control will handle the parsing, layout, and scrollable view of HTML documents and inline images including GIF, JPEG, BMP, and XBM.

The HTML control also lets you implement an HTML viewer, with or without automatic network retrieval of HTML documents. There is built-in document retrieval for HTTP and File URLs. The HTML Control can also be used as a non-visual HTML parser to analyze or process HTML documents.

Properties:

BackColor	Defines the default background color. May be overridden by the DocBackColor property, if such a document color is present and the UseDocColors property is True.
BackImage	URL of an image to be used as the background image of the document. May be overridden by the background image of the document () if this attribute is present and the UseDocColors property is True. The background image is tiled to fill the view area of the control window.
BaseURL	URL of the element of the current document, used for relative URL resolution. If no element exists in the document, this property is the same as the URL property.
DeferRetrieval	Indicates whether retrieval of embedded objects should be deferred until explicitly requested. The user can set this property to turn inline retrieval of embedded documents off or on. If you are implementing caching, you will normally leave this property set to False so that cached documents are always displayed inline.
DocBackColor	Document background color. This property corresponds to the BGCOLOR attribute of the BODY tag. If this attribute is not present, HTML defaults to the value of the BackColor property.
DocForeColor	Document foreground (text) color. This property corresponds to the TEXT attribute of the BODY tag. If this attribute is not present, HTML defaults to the value of the ForeColor property.
DocInput	Object describing input information for the main document being transferred. The DocInput object provides a more powerful interface than the basic capabilities of the RequestDoc method. However, you can use the basic functions of the control without knowledge or use of the DocInput object.
DocLinkColor	This property corresponds to the LINK attribute of

	the BODY tag. If this attribute is not present, HTML defaults to the value of the LinkColor property.
DocOutput	Object describing output information when submitting form data. The DocOutput object provides a more powerful interface than the basic capabilities of the RequestSubmit method. However, you can use the basic functions of the control without knowledge or use of the DocInput object.
DocVisitedColor	Document visited link color. This property corresponds to the VLINK attribute of the BODY tag. If this attribute is not present, HTML defaults to the value of the VisitedColor property.
ElemNotification	Indicates whether the DoNewElement event should be activated during HTML parsing. You can set this property to True when using the HTML Control as a (visual or nonvisual) parser.
FixedFont	Font for fixed-width text.
Font	Font for regular text.
ForeColor	Default foreground (text) color. This property may be overridden by the DocForeColor property if such a document color is present and the UseDocColors property is True.
Forms	A collection of the forms contained in the HTML page. This property may be indexed directly to call the default Item method.
Heading1Font	Font for heading level 1 text (<H1> elements).
Heading2Font	Font for heading level 2 text (<H2> elements).
Heading3Font	Font for heading level 3 text (<H3> elements).
Heading4Font	Font for heading level 4 text (<H4> elements).
Heading5Font	Font for heading level 5 text (<H5> elements).
Heading6Font	Font for heading level 6 text (<H6> elements).
LayoutDone	Indicates whether the layout phase is complete. This property is set to False when document retrieval starts, and set to True when layout (placement of items on the page) of the main document is complete.
LinkColor	Default link color. This property may be overridden by the DocLinkColor property if such a document color is present and the UseDocColors property is True.
ParseDone	Indicates whether the parsing phase is complete. This property is set to False when document retrieval starts, and set to True when parsing of the main document is complete.

Redraw	Indicates whether drawing should occur as data changes or the window is scrolled. To make changes and avoid flickering (redrawing when each change is made), set the Redraw property to False, make the changes, and then set it back to True. When Redraw is set to True, the window will be redrawn.
RequestURL	URL string identifying the new document requested. You can specify this property by calling RequestDoc. The property is set by the control during default processing for the DoRequestDoc event.
RetainSource	Indicates whether source text should be retained and available via the SourceText property. This property may be set to False to save memory when you do not need the source text of the main document.
RetrieveBytesDone	Completed byte size of the objects being retrieved. This property is zero if no retrieval is in progress.
RetrieveBytesTotal	Total byte size of the objects to be retrieved, including embedded objects and the document itself. If DeferRetrieval is set to True,RetrieveBytesTotal does not include embedded objects. This value can change during retrieval as object sizes are determined. This property is zero if no retrieval is in progress.
SourceText	Contains the source text of the main document. This property will be empty if the RetainSource property is False or if no main document has been retrieved.
Timeout	Time-out interval (in seconds) for initiating the request for documents. The Timeout event is activated if no data is received within timeout. Although the Timeout value applies to all document retrieval, the Timeout event is activated only for the main document, not for embedded documents. Event is an integer value that determines the type of Timeout event that will be enabled.
TotalHeight	Total height of the document in pixels. This property reflects the total height of the document, including the area that may not be visible because the view is smaller than the document. This property is updated as parsing and layout of the HTML document occurs. Its value is final when the EndRetrieval event is activated.
TotalWidth	Total width of the document in pixels. This property reflects the total width of the document, including the area that may not be visible because the view is smaller than the document. This property is updated as parsing and layout of the HTML document occurs. Its value is final when the EndRetrieval event is activated.

UnderlineLinks	Indicates whether links should be underlined.
URL	URL string identifying the current main document. This property is set by the control from the URLRequest property when document retrieval has successfully started and the BeginRetrieval event is activated.

Methods:

Cancel	Used to terminate document retrieval (including embedded documents), and optionally output a message at the end of the partially retrieved HTML page.
RequestAllEmbedded	Requests retrieval of all embedded documents via the DoRequestEmbedded event. This method is used in conjunction with the DeferRetrieval property to control inline display of embedded documents.
RequestDoc	Requests retrieval of a new main document identified by the URL. When RequestDoc is called, the DoRequestDoc event is activated to determine the DocStream to be used for retrieval. The RequestURL property will then be set to the URL parameter specified. The URL property will not be updated until retrieval is successfully underway and the BeginRetrieval event is activated.

Events:

OnBeginRetrieval	This event is activated when document retrieval begins.
OnDocInput	A DocInput related event that indicates the input data has been transferred. The DocInput event can be used in its basic form for notification of transfer progress.
OnDocOutput	A DocOutput related event indicating that output data has been transferred. The DocOutput event can be used in its basic form to notify the user of transfer progress.
OnDoNewElement	The event is activated during HTML parsing when a new element is added.
OnDoRequuestDoc	The event is activated when the user chooses a link to a different URL or when the RequestDoc method is called.
OnDoRequuestEmbed	The event is activated when an embedded document, such as an image is to be retrieved for inline display.
OnDoRequestSubmit	The event is activated when the user selects form submission, or when the RequestSubmit method of the Form is called.

OnEndRetrieval	The event is activated when document retrieval, including embedded documents to be displayed inline, is complete.
OnError	This event is activated when an error occurs in background processing.
OnLayoutComplete	The event is activated when layout of the HTML document is complete. Embedded document retrieval may not be complete, however, at least the size of each embedded document and the position of all elements has been determined.
OnParseComplete	The event is activated when parsing of the HTML document is complete.
OnTimeout	The event is activated after no data has been received within the time specified in the TimeOut property.
OnUpdateRetrieval	The event is activated periodically as the document and embedded objects are retrieved. The RetrieveBytesTotal and RetrieveBytesDone properties can be queried at the time this event is activated to update a progress bar.

As Declared in ISP.PAS:

```
{ THTML }

THTMLError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
THTMLDocInput = procedure(Sender: TObject; const DocInput: Variant) of
object;
THTMLDocOutput = procedure(Sender: TObject; const DocOutput: Variant) of
object;
THTMLDoRequestDoc = procedure(Sender: TObject; const URL: string; const
Element, DocInput: Variant; var EnableDefault: TOleBool) of object;
THTMLDoRequestEmbedded = procedure(Sender: TObject; const URL: string;
const Element, DocInput: Variant; var EnableDefault: TOleBool) of object;
THTMLDoRequestSubmit = procedure(Sender: TObject; const URL: string; const
Form, DocOutput: Variant; var EnableDefault: TOleBool) of object;
THTMLDoNewElement = procedure(Sender: TObject; const ElemType: string;
EndTag: TOleBool; const Attrs: Variant; const Text: string; var
EnableDefault: TOleBool) of object;

THTML = class(TOleControl)
  private
    FOnError: THTMLError;
    FOnDocInput: THTMLDocInput;
    FOnDocOutput: THTMLDocOutput;
    FOnParseComplete: TNotifyEvent;
    FOnLayoutComplete: TNotifyEvent;
    FOnTimeout: TNotifyEvent;
    FOnBeginRetrieval: TNotifyEvent;
    FOnUpdateRetrieval: TNotifyEvent;
    FOnEndRetrieval: TNotifyEvent;
    FOnDoRequestDoc: THTMLDoRequestDoc;
    FOnDoRequestEmbedded: THTMLDoRequestEmbedded;
    FOnDoRequestSubmit: THTMLDoRequestSubmit;
    FOnDoNewElement: THTMLDoNewElement;
```

```
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure RequestDoc(const URL: string); stdcall;
    procedure RequestAllEmbedded; stdcall;
    procedure Cancel(const Message: Variant); stdcall;
    property DocInput: Variant index 1002 read GetVariantProp;
    property DocOutput: Variant index 1003 read GetVariantProp;
    property URL: string index 1001 read GetStringProp;
    property RequestURL: string index 2 read GetStringProp;
    property BaseURL: string index 3 read GetStringProp;
    property Forms: Variant index 4 read GetVariantProp;
    property TotalWidth: Integer index 5 read GetIntegerProp;
    property TotalHeight: Integer index 6 read GetIntegerProp;
    property RetrieveBytesTotal: Integer index 7 read GetIntegerProp;
    property RetrieveBytesDone: Integer index 8 read GetIntegerProp;
    property ParseDone: TOleBool index 9 read GetOleBoolProp;
    property LayoutDone: TOleBool index 10 read GetOleBoolProp;
    property SourceText: string index 14 read GetStringProp;
    property DocBackColor: TColor index 23 read GetColorProp;
    property DocForeColor: TColor index 24 read GetColorProp;
    property DocLinkColor: TColor index 25 read GetColorProp;
    property DocVisitedColor: TColor index 26 read GetColorProp;
    property Errors: Variant index 508 read GetVariantProp;
  published
    property Align;
    property ParentColor;
    property ParentFont;
    property TabStop;
    property DragCursor;
    property DragMode;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property TabOrder;
    property Visible;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnStartDrag;
    property OnClick;
    property OnDblClick;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property DeferRetrieval: TOleBool index 11 read GetOleBoolProp write
        SetOleBoolProp stored False;
    property ViewSource: TOleBool index 12 read GetOleBoolProp write
        SetOleBoolProp stored False;
    property RetainSource: TOleBool index 13 read GetOleBoolProp write
        SetOleBoolProp stored False;
    property ElemNotification: TOleBool index 15 read GetOleBoolProp write
        SetOleBoolProp stored False;
    property Timeout: Integer index 507 read GetIntegerProp write
        SetIntegerProp stored False;
```

```
  property Redraw: TOleBool index 17 read GetOleBoolProp write
      SetOleBoolProp stored False;
  property UnderlineLinks: TOleBool index 18 read GetOleBoolProp write
      SetOleBoolProp stored False;
  property UseDocColors: TOleBool index 19 read GetOleBoolProp write
      SetOleBoolProp stored False;
  property BackImage: string index 20 read GetStringProp write
      SetStringProp stored False;
  property BackColor: TColor index -501 read GetColorProp write
      SetColorProp stored False;
  property ForeColor: TColor index -513 read GetColorProp write
      SetColorProp stored False;
  property LinkColor: TColor index 21 read GetColorProp write
      SetColorProp stored False;
  property VisitedColor: TColor index 22 read GetColorProp write
      SetColorProp stored False;
  property Font: Variant index -512 read GetVariantProp write
      SetVariantProp stored False;
  property FixedFont: Variant index 27 read GetVariantProp write
      SetVariantProp stored False;
  property Heading1Font: Variant index 28 read GetVariantProp write
      SetVariantProp stored False;
  property Heading2Font: Variant index 29 read GetVariantProp write
      SetVariantProp stored False;
  property Heading3Font: Variant index 30 read GetVariantProp write
      SetVariantProp stored False;
  property Heading4Font: Variant index 31 read GetVariantProp write
      SetVariantProp stored False;
  property Heading5Font: Variant index 32 read GetVariantProp write
      SetVariantProp stored False;
  property Heading6Font: Variant index 33 read GetVariantProp write
      SetVariantProp stored False;
  property OnError: THTMLError read FOnError write FOnError;
  property OnDocInput: THTMLDocInput read FOnDocInput write FOnDocInput;
  property OnDocOutput: THTMLDocOutput read FOnDocOutput write
      FOnDocOutput;
  property OnParseComplete: TNotifyEvent read FOnParseComplete write
      FOnParseComplete;
  property OnLayoutComplete: TNotifyEvent read FOnLayoutComplete write
      FOnLayoutComplete;
  property OnTimeout: TNotifyEvent read FOnTimeout write FOnTimeout;
  property OnBeginRetrieval: TNotifyEvent read FOnBeginRetrieval write
      FOnBeginRetrieval;
  property OnUpdateRetrieval: TNotifyEvent read FOnUpdateRetrieval write
      FOnUpdateRetrieval;
  property OnEndRetrieval: TNotifyEvent read FOnEndRetrieval write
      FOnEndRetrieval;
  property OnDoRequestDoc: THTMLDoRequestDoc read FOnDoRequestDoc write
      FOnDoRequestDoc;
  property OnDoRequestEmbedded: THTMLDoRequestEmbedded read
      FOnDoRequestEmbedded write FOnDoRequestEmbedded;
  property OnDoRequestSubmit: THTMLDoRequestSubmit read
      FOnDoRequestSubmit write FOnDoRequestSubmit;
  property OnDoNewElement: THTMLDoNewElement read FOnDoNewElement write
      FOnDoNewElement;
end;
```

Example:

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, Menus, ComCtrls, OleCtrls, ISP, Buttons;

type
  TForm1 = class(TForm)
    HTML1: THTML;
    StatusBar1: TStatusBar;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Panel1: TPanel;
    Bevel1: TBevel;
    URLs: TComboBox;
    GoButton: TSpeedButton;
    View1: TMenuItem;
    DocumentSource1: TMenuItem;
    CancelBtn: TSpeedButton;
    Label1: TLabel;
    procedure Exit1Click(Sender: TObject);
    procedure GoButtonClick(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure DocumentSource1Click(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
    procedure HTML1BeginRetrieval(Sender: TObject);
    procedure HTML1EndRetrieval(Sender: TObject);
    procedure URLsKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses DocSrc;

{$R *.DFM}

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.GoButtonClick(Sender: TObject);
begin
  if URLs.Items.IndexOf(URLs.Text) = -1 then URLs.Items.Add(URLs.Text);
  HTML1.RequestDoc(URLs.text);
  Statusbar1.Panels[0].Text := HTML1.RequestURL;
end;
```

```
procedure TForm1.About1Click(Sender: TObject);
begin
  HTML1.AboutBox;
end;

procedure TForm1.DocumentSource1Click(Sender: TObject);
begin
  with DocSourceFrm do
  begin
    Show;
    Memo1.Lines.Clear;
    Memo1.Lines.Add(HTML1.SourceText);
    Memo1.SelStart := 0;
    SendMessage(Memo1.Handle, EM_ScrollCaret, 0, 0);
  end;
end;

procedure TForm1.CancelBtnClick(Sender: TObject);
begin
  HTML1.Cancel('test');
  CancelBtn.Enabled := False;
end;

procedure TForm1.HTML1BeginRetrieval(Sender: TObject);
begin
  CancelBtn.Enabled := True;
end;

procedure TForm1.HTML1EndRetrieval(Sender: TObject);
begin
  CancelBtn.Enabled := False;
end;

procedure TForm1.URLsKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = VK_Return then GoButtonClick(nil);
end;

end.
```

HTTP - Hypertext Transport Protocol

The HTTP (Hypertext Transport Protocol) Control implements the HTTP Protocol Client based on the HTTP specification. This control lets you directly retrieve HTTP documents if no browsing or image processing is necessary.

It can be used by developers who implement HTML browsers or other services that need access to HTTP. For example, the HTML Control internally instantiates this object and uses it for HTTP transactions.

The HTTP Control uses a number of methods to retrieve or send (post) a document. It can retrieve MIME information about the document from the Headers collection property.

Properties:

Busy	Indicates a command is in progress.
DocInput	Object describing input information for the main document being transferred. The DocInput object provides a more powerful interface than the basic capabilities of the RequestDoc method. However, you can use the basic functions of the control without knowledge or use of the DocInput object.
DocOutput	Object describing output information when submitting form data. The DocOutput object provides a more powerful interface than the basic capabilities of the RequestSubmit method. However, you can use the basic functions of the control without knowledge or use of the DocInput object.
Document	Identifies the target document. The Document property can be used with RemoteHost to identify the URL. It can also be used instead of URL.
EnableTimer	Boolean property to enable timer for the specified event. Value is specified in the TimeOut property.
Errors	A collection of errors that can be accessed for details about the last error that occurred. This collection should be used within an Error event if information passed through the Error event is not sufficient.
Method	Method used to retrieve or post (send) the document. prcGet = 1 Get method request the whole document. prcHead = 2 Head method requests only the headers of a document. prcPost = 3 Post method posts the whole document to the server as a sub-ordinate of the document specified by the URL. prcPut = 4 Put method puts the whole document to the server. The document replaces an existing document specified by the URL.

NotificationMode	Determines when notification is issued for incoming data. Notification can also be suspended. 0 = COMPLETE: notification is provided when there is a complete response. 1 = CONTINUOUS: an event is repeatedly activated when new data arrives from the connection.
ProtocolState	This property specifies the current state of the protocol. Constants defined for enum types of ProtocolState property are: ftpBase (default)= 0 (Default) - the state before connection server is established. ftpAuthorization = 1 - authorization is performed. ftpTransaction = 2 - Authorization successful. The client has successfully identified itself to the FTP server.
ProtocolStateString	String representation of ProtocolState.
RemoteHost	The remote machine to connect to if the remoteHost parameter in the Connect method is missing. You can either provide a host name or an IP address string in dotted format. For example, 127.0.0.1.
RemotePort	The remote port number to which to connect.
ReplyCode	The value of the reply code is a protocol specific number that determines the result of the last request, as returned in the ReplyString property. See RFC 959 for Valid reply codes.
ReplyString	Lists the last reply string sent by the FTP Server to the client as a result of a request. This string contains both a number code and a status string that the server creates for the last command.
State	This property specifies the connection state of the control. prcConnecting = 1 - Connecting. Connect has been requested, waiting for connect acknowledge. prcResolvingHost = 2 - Resolving Host. Occurs when RemoteHost is in name format rather than dot-delimited IP format. prcHostResolved = 3 - Resolved the host. Occurs only if ResolvingHost state has been entered previously. prcConnected = 4 - Connection established. prcDisconnecting = 5 - Connection closed. Disconnect has been initiated. prcDisconnected = 6 - Initial state when protocol object is instantiated, before Connect has been initiated, after a Connect attempt failed or after Disconnect performed.
StateString	A string representation of State.
TimeOut	Timeout value for the specified event.

URL	URL string identifying the current document being transferred. URL format is: HTTP://host:port/documentnameandpath

Methods:

Cancel	Cancels a pending request.
Connect	Initiates a Connect request. The control calls the OnStateChanged event if a connection is established
GetDoc	A DocOutput related method that requests retrieval of a document identified by a URL. The GetDoc method in FTP means retrieving a file from the server.
PerformRequest	Initiates a request method to retrieve a document. If no parameters are specified, properties Document, HostName, RemotePort and Method are used for the retrieval. This method is similar to GetDoc, except it uses a different set of parameters.
SendDoc	A DocInput related method that requests sending a document identified by a URL. The SendDoc method in FTP means putting a file on the server.

Events:

OnBusy	This event is activated when a command is in progress or when a command has completed.
OnCancel	This event is activated after a cancellation request has been completed and satisfied. After this event the object's state changes to idle.
OnDocInput	A DocInput related event that indicates the input data has been transferred. The DocInput event can be used in its basic form for notification during transfer.
OnDocOutput	A DocOutput related event indicating that output data has been transferred. The DocOutput event can be used in its basic form for notification during transfer.
OnError	This event is activated when an error occurs in background processing. See the online help for error codes.
OnLog	This event is activated when the Logging property is set to TRUE.
OnProtocolStateChanged	This event is activated whenever the protocol state changes.
OnStateChanged	This event is activated whenever the state of the transport state changes.

OnTimeout	This event is activated when the timer for the specified event expires. See Timeout property for pre-defined events.

As Declared in ISP.PAS:

```
{ THTTP }

THTTPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
THTTPTimeout = procedure(Sender: TObject; event: Smallint; var Continue:
TOleBool) of object;
THTTPStateChanged = procedure(Sender: TObject; State: Smallint) of object;
THTTPProtocolStateChanged = procedure(Sender: TObject; ProtocolState:
Smallint) of object;
THTTPBusy = procedure(Sender: TObject; isBusy: TOleBool) of object;
THTTPDocInput = procedure(Sender: TObject; const DocInput: Variant) of
object;
THTTPDocOutput = procedure(Sender: TObject; const DocOutput: Variant) of
object;

THTTP = class(TOleControl)
  private
    FOnError: THTTPError;
    FOnTimeout: THTTPTimeout;
    FOnCancel: TNotifyEvent;
    FOnStateChanged: THTTPStateChanged;
    FOnProtocolStateChanged: THTTPProtocolStateChanged;
    FOnBusy: THTTPBusy;
    FOnLog: TNotifyEvent;
    FOnDocInput: THTTPDocInput;
    FOnDocOutput: THTTPDocOutput;
    function Get_Timeout(event: Smallint): Integer; stdcall;
    procedure Set_Timeout(event: Smallint; Value: Integer); stdcall;
    procedure Set_EnableTimer(event: Smallint; Value: TOleBool); stdcall;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure Cancel; stdcall;
    procedure SendDoc(const URL, Headers, InputData, InputFile, OutputFile:
        Variant); stdcall;
    procedure GetDoc(const URL, Headers, OutputFile: Variant); stdcall;
    property State: Smallint index 503 read GetSmallintProp;
    property ProtocolState: Smallint index 504 read GetSmallintProp;
    property ReplyString: string index 505 read GetStringProp;
    property ReplyCode: Integer index 506 read GetIntegerProp;
    property Errors: Variant index 508 read GetVariantProp;
    property Busy: TOleBool index 509 read GetOleBoolProp;
    property StateString: string index 511 read GetStringProp;
    property ProtocolStateString: string index 512 read GetStringProp;
    property DocInput: Variant index 1002 read GetVariantProp;
    property DocOutput: Variant index 1003 read GetVariantProp;
    property Timeout[event: Smallint]: Integer read Get_Timeout write
        Set_Timeout;
    property EnableTimer[event: Smallint]: TOleBool write Set_EnableTimer;
  published
    property RemoteHost: string index 0 read GetStringProp write
        SetStringProp stored False;
    property RemotePort: Integer index 502 read GetIntegerProp write
        SetIntegerProp stored False;
```

```
    property NotificationMode: Smallint index 510 read GetSmallintProp
        write SetSmallintProp stored False;
    property Logging: TOleBool index 514 read GetOleBoolProp write
        SetOleBoolProp stored False;
    property Document: string index 10 read GetStringProp write
        SetStringProp stored False;
    property Method: TOleEnum index 11 read GetOleEnumProp write
        SetOleEnumProp stored False;
    property URL: string index 1001 read GetStringProp write SetStringProp
        stored False;
    property OnError: THTTPError read FOnError write FOnError;
    property OnTimeout: THTTPTimeout read FOnTimeout write FOnTimeout;
    property OnCancel: TNotifyEvent read FOnCancel write FOnCancel;
    property OnStateChanged: THTTPStateChanged read FOnStateChanged write
        FOnStateChanged;
    property OnProtocolStateChanged: THTTPProtocolStateChanged read
        FOnProtocolStateChanged write FOnProtocolStateChanged;
    property OnBusy: THTTPBusy read FOnBusy write FOnBusy;
    property OnLog: TNotifyEvent read FOnLog write FOnLog;
    property OnDocInput: THTTPDocInput read FOnDocInput write FOnDocInput;
    property OnDocOutput: THTTPDocOutput read FOnDocOutput write
        FOnDocOutput;
  end;
```

Example:

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, Menus, ComCtrls, OleCtrls, ISP, Buttons, Ole2;

type
  TForm1 = class(TForm)
    StatusBar1: TStatusBar;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Panel1: TPanel;
    Bevel1: TBevel;
    URLs: TComboBox;
    GoButton: TSpeedButton;
    CancelBtn: TSpeedButton;
    Label1: TLabel;
    HTTP1: THTTP;
    Memo1: TMemo;
    procedure Exit1Click(Sender: TObject);
    procedure GoButtonClick(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
    procedure HTML1BeginRetrieval(Sender: TObject);
    procedure HTML1EndRetrieval(Sender: TObject);
    procedure URLsKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure HTTP1DocOutput(Sender: TObject; const DocOutput: Variant);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
  Data: String;

implementation
{$R *.DFM}

{ TSimpleHTMLParser }

type

  TToken = (etEnd, etSymbol, etLineEnd, etHTMLTag);

  TSimpleHTMLParser = class
  private
    FText: string;
    FSourcePtr: PChar;
    FTokenPtr: PChar;
    FTokenString: string;
    FToken: TToken;
    procedure NextToken;
    procedure NextSymbol;
    function TokenSymbolIs(const S: string): Boolean;
    function TokenHTMLTagIs(const S: string): Boolean;
  public
    constructor Create(const Text: string);
  end;

constructor TSimpleHTMLParser.Create(const Text: string);
begin
  FText := Text;
  FSourcePtr := PChar(Text);
  NextToken;
end;

procedure TSimpleHTMLParser.NextToken;
var
  P, TokenStart: PChar;
  StrBuf: array[0..255] of Char;
begin
  FTokenString := '';
  P := FSourcePtr;
  while (P^ <> #0) and (P^ <= ' ') do Inc(P);
  FTokenPtr := P;
  case P^ of
    '<':
      begin
        Inc(P);
        TokenStart := P;
        while (P^ <> '>') and (P^ <> #0) do Inc(P);
        SetString(FTokenString, TokenStart, P - TokenStart);
        FToken := etHTMLTag;
        Inc(P);
      end;
    #13: FToken := etLineEnd;
    #0: FToken := etEnd;
  else
    begin
      TokenStart := P;
      Inc(P);
      while not (P^ in ['<', #0, #13,#10]) do Inc(P);
```

```
      SetString(FTokenString, TokenStart, P - TokenStart);
      FToken := etSymbol;
    end;
  end;
  FSourcePtr := P;
end;

procedure TSimpleHTMLParser.NextSymbol;
begin
  while (FToken <> etEnd) do
  begin
    NextToken;
    if FToken = etSymbol then break;
  end;
end;

function TSimpleHTMLParser.TokenSymbolIs(const S: string): Boolean;
begin
  Result := (FToken = etSymbol) and (CompareText(FTokenString, S) = 0);
end;

function TSimpleHTMLParser.TokenHTMLTagIs(const S: string): Boolean;
begin
  Result := (FToken = etHTMLTag) and ((CompareText(FTokenString, S) = 0) or
    (Pos(S, FTokenString) = 1));
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.GoButtonClick(Sender: TObject);
var
  a,b: variant;
begin
  Memo1.Lines.Clear;
  if URLs.Items.IndexOf(URLs.Text) = -1 then URLs.Items.Add(URLs.Text);
  HTTP1.GetDoc(URLs.text, a, b);
  Statusbar1.Panels[0].Text := HTTP1.URL;
end;

procedure TForm1.CancelBtnClick(Sender: TObject);
begin
  HTTP1.Cancel;
  CancelBtn.Enabled := False;
end;

procedure TForm1.HTML1BeginRetrieval(Sender: TObject);
begin
  CancelBtn.Enabled := True;
end;

procedure TForm1.HTML1EndRetrieval(Sender: TObject);
begin
  CancelBtn.Enabled := False;
end;

procedure TForm1.URLsKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
```

```
  if Key = VK_Return then GoButtonClick(nil);
end;

procedure TForm1.HTTP1DocOutput(Sender: TObject; const DocOutput: Variant);
var
  S: String;
  i: integer;
  MsgNo, Header: String;
  Parser: TSimpleHTMLParser;
  ALine: String;
begin
  Statusbar1.Panels[2].Text := Format('Bytes:
        %s',[DocOutput.BytesTransferred]);
  case DocOutput.State of
    icDocBegin:
      begin
        Memo1.Lines.Clear;
        Data := '';
      end;
    icDocData:
      begin
        DocOutput.GetData(S, VT_BSTR);
        Data := Data + S;
      end;
    icDocEnd:
      begin
        { Now remove all the HTML tags and only display the text }
        Parser := TSimpleHTMLParser.Create(Data);
        ALine := '';
        while Parser.FToken <> etEnd do
        begin
          case Parser.FToken of
            etHTMLTag:
              begin
                if Parser.TokenHTMLTagIs('BR') then
                  ALine := ALine + #13#10;
                if Parser.TokenHTMLTagIs('P') then
                  ALine := ALine + #13#10#13#10;
              end;
            etSymbol: ALine := ALine + ' ' + Parser.FTokenString;
            etLineEnd:
              begin
                Memo1.Lines.Add(ALine);
                ALine := '';
              end;
          end;
          Parser.NextToken;
        end;
        Memo1.Lines.Add(ALine);
        Memo1.SelStart := 0;
        SendMessage(Memo1.Handle, EM_ScrollCaret, 0, 0);
      end;
  end;
  Refresh;
end;

end.
```

NNTP - Networking News Transfer Protocol

The Networking News Transfer Protocol (NNTP) Client Control implements the basic client NNTP Protocol as specified by RFC 977, Network News Transfer Protocol. THE NNTP Control also implements NNTP extension commands as documented in the Internet-Draft on Common NNTP Extensions. The NNTP Control provides a reusable component that allows applications to access NNTP news servers. It provides news reading and posting capabilities.

Properties:

ArticleNumbersSupported	If True, the GetArticleNumbers method may be used to retrieve a list of article numbers for a newsgroup. This property has no meaning before the connection to the server has been established.
Busy	Indicates a command is in progress.
DocInput	Object describing input information for the document being transferred.
DocOutput	Object describing output information for the document being transferred.
EnableTimer	Boolean property to enable timer for the specified event. Value is specified in the TimeOut property.
Errors	A collection of errors that can be accessed for details about the last error that occurred. This collection should be used within an Error event if information passed through the Error event is not sufficient.
LastUpdate	The default value used by the GetAdministrationFile and ListNewGroups methods.
NotificationMode	Determines when notification is issued for incoming data. Notification can also be suspended. 0 = COMPLETE: notification is provided when there is a complete response. 1 = CONTINUOUS: an event is repeatedly activated when new data arrives from the connection.
OverviewSupported	If True, the GetOverviewFormat and GetOverview methods may be used to retrieve header information stored in the server's overview database. This property has no meaning before the connection to the server has been established.
PostingAllowed	If True, the current NNTP server allows posting of news articles. This property has no meaning before the connection to the server has been established.
ProtocolState	This property specifies the current state of the protocol. Constants defined for enum types of ProtocolState property are: prcBase = 0 Base state before connection to server is established.

prcTransaction = 1 Connection to server is established. This is the valid state for calling methods on the control.

ProtocolStateString — String representation of ProtocolState.

RemoteHost — The remote machine to connect to if the remoteHost parameter in the Connect method is missing. You can either provide a host name or an IP address string in dotted format. For example, 127.0.0.1.

RemotePort — The remote port number to which to connect.

ReplyCode — The value of the reply code is a protocol specific number that determines the result of the last request, as returned in the ReplyString property. See RFC 977 for a list of valid reply codes.

ReplyString — Lists the last reply string sent by the FTP Server to the client as a result of a request. This string contains both a number code and a status string that the server creates for the last command.

State — This property specifies the connection state of the control.

prcConnecting = 1 - Connecting. Connect has been requested, waiting for connect acknowledge.
prcResolvingHost = 2 - Resolving Host. Occurs when RemoteHost is in name format rather than dot-delimited IP format.
prcHostResolved = 3 - Resolved the host. Occurs only if ResolvingHost state has been entered previously.
prcConnected = 4 - Connection established.
prcDisconnecting = 5 - Connection closed. Disconnect has been initiated.
prcDisconnected = 6 - Initial state when protocol object is instantiated, before Connect has been initiated, after a Connect attempt failed or after Disconnect performed.

StateString — A string representation of State.

TimeOut — Timeout value for the specified event.

URL — URL string identifying the current document being transferred. The valid URL formats are:

news:<newsgroupname>
news:<messageid>

Methods:

Cancel — Cancels a pending request.

Connect — Initiates a Connect request. The control calls the StateChanged event if a connection is established. Optional arguments to this method override the values from corresponding RemoteHost and RemotePort properties. The values of the properties will

not change. If no argument is given, the values from the properties will be used to establish the connection.

GetAdministrationFile — Sends the NNTP XMOTD command to the server. This command retrieves the news server administrator's information if the information is newer than the value of lastUpdate.

GetArticleByArticleNumber — Sends the NNTP ARTICLE command with articleNumber to the NNTP server. Upon successful completion, this method causes the DocOutput event to be activated.

GetArticleByMessageID — Sends the NNTP ARTICLE command with articleID to the server. When this method reaches a successful completion, the DocInput event is activated.

GetArticleHeaders — Sends the NNTP XHDR command to the server. Upon successful completion, this method causes the DocOutput event to be activated.

GetArticleNumbers — Sends the NNTP command LISTGROUP to the server. Upon successful completion, this method causes the DocOutput event to be activated. Use the ArticleNumbersSupported property after connection to determine if the current NNTP server supports this command.

GetBodyByArticleNumber — Sends the NNTP BODY command with articleNumber to the NNTP server. Upon successful completion, this method causes the DocOutput event to be activated.

GetBodyByMessageID — Sends the NNTP BODY command with messageID to the server. Upon successful completion, this method causes the DocOutput event to be activated.

GetDoc — A DocOutput related method that requests retrieval of a document identified by a URL. The GetDoc method in NNTP means retrieving an article from the NNTP server. The URL and (for some controls) Headers are used as inputs specifying which document is to be retrieved. The OutputFile argument indicates where the retrieved document should be written locally.

GetHeaderByArticleNumber — Sends the NNTP HEAD command with messageNumber to the NNTP server. Upon successful completion, this method causes the DocOutput event to be activated.

GetHeaderByMessageID — Sends the NNTP HEAD command with messageID to the server. Upon successful completion, this method causes the DocOutput event to be activated.

GetOverview — Sends the XOVER command to the server. Use the OverSupported property after connection to determine if the current NNTP server supports this command. When this method reaches a successful completion, the DocInput event is activated. The

	XOVER command returns information from the overview database for the article(s) specified.
GetOverviewFormat	Sends the LIST OVERVIEW.FMT command to the server. Use the OverViewSupported property after connection to determine if the current NNTP server supports this command. When this method reaches a successful completion, the DocInput event is activated.
GetStatByArticleNumber	Sends the NNTP STAT command with articleNumber to the NNTP server. When this method reaches a successful completion, the StatArticle event is activated.
ListGroupDescriptions	Sends the NNTP LIST NEWSGROUPS command to the server. Upon successful completion, this method causes the DocOutput event to be activated.
ListGroups	Sends NNTP LIST command to the server. The server responds with a list of all news groups. Upon successful completion, this method causes the DocOutput event to be activated.
ListNewGroups	Sends NNTP NEWGROUPS command to server. Upon successful completion, this method causes the DocOutput event to be activated.
Quit	Sends NNTP QUIT command and disconnects from the NNTP server. When this method reaches a successful completion, the StateChanged event is activated.
SelectGroup	Sends NNTP GROUP command to the server. On successful completion, the SelectGroup event is activated.
SendDoc	A DocInput related method that requests sending a document identified by a URL. The SendDoc method in NNTP means posting an article to the NNTP server.
SetLastArticle	Sends NNTP LAST command to the server. On successful completion, the LastArticle event is activated.
SetNextArticle	Sends NNTP NEXT command to the server. On successful completion, the NextArticle event is activated.

Events:

OnAuthenticateRequest	This event is activated when the connected NNTP server requests authentication. If the UserID and Password arguments are specified, their values are used instead of the UserID and Password properties.
OnAuthenticateResponse	This event is activated when an authentication response is received from the server.
OnBanner	This event is activated when the server responds with its sign-on banner after a connection is established.

OnBusy	This event is activated when a command is in progress or when a command has completed.
OnCancel	This event is activated after a cancellation request has been completed and satisfied. After this event the object's state changes to idle.
OnDocInput	A DocInput related event that indicates the input data has been transferred. The DocInput event can be used in its basic form for notification during transfer.
OnDocOutput	A DocOutput related event indicating that output data has been transferred. The DocOutput event can be used in its basic form for notification during transfer.
OnError	This event is activated when an error occurs in background processing (for example, failed to connect or failed to send or receive in the background). NNTP Error Codes : The following error codes apply only to the NNTP ActiveX Control. Error Code — Error Message 2203 — NNTP server does not allow posting.
OnLastArticle	This event is activated after a successful completion of the LastArticle method.
OnNextArticle	This event is activated after a successful completion of the NextArticle method.
OnProtocolStateChanged	This event is activated whenever the protocol state changes.
OnSelectGroup	This event is activated after a successful completion of the SelectGroup method.
OnStatArticle	This event is activated after a successful completion of the GetStatByArticleNumber method.
OnStateChanged	This event is activated whenever the state of the transport state changes.
OnTimeout	This event is activated when the timer for the specified event expires. See Timeout property for pre-defined events.

As Declared in ISP.PAS:

```
{ TNNTP }

TNNTPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
TNNTPTimeout = procedure(Sender: TObject; event: Smallint; var Continue:
TOleBool) of object;
TNNTPStateChanged = procedure(Sender: TObject; State: Smallint) of object;
```

```
TNNTPProtocolStateChanged = procedure(Sender: TObject; ProtocolState:
Smallint) of object;
TNNTPBusy = procedure(Sender: TObject; isBusy: TOleBool) of object;
TNNTPDocInput = procedure(Sender: TObject; const DocInput: Variant) of
object;
TNNTPDocOutput = procedure(Sender: TObject; const DocOutput: Variant) of
object;
TNNTPAuthenticateRequest = procedure(Sender: TObject; var UserId, Password:
string) of object;
TNNTPAuthenticateResponse = procedure(Sender: TObject; Authenticated:
TOleBool) of object;
TNNTPBanner = procedure(Sender: TObject; const Banner: string) of object;
TNNTPSelectGroup = procedure(Sender: TObject; const groupName: string;
firstMessage, lastMessage, msgCount: Integer) of object;
TNNTPNextArticle = procedure(Sender: TObject; articleNumber: Integer; const
messageID: string) of object;
TNNTPlastArticle = procedure(Sender: TObject; articleNumber: Integer; const
messageID: string) of object;
TNNTPArticleStatus = procedure(Sender: TObject; articleNumber: Integer;
const messageID: string) of object;

TNNTP = class(TOleControl)
  private
    FOnError: TNNTPError;
    FOnTimeout: TNNTPTimeout;
    FOnCancel: TNotifyEvent;
    FOnStateChanged: TNNTPStateChanged;
    FOnProtocolStateChanged: TNNTPProtocolStateChanged;
    FOnBusy: TNNTPBusy;
    FOnLog: TNotifyEvent;
    FOnDocInput: TNNTPDocInput;
    FOnDocOutput: TNNTPDocOutput;
    FOnAuthenticateRequest: TNNTPAuthenticateRequest;
    FOnAuthenticateResponse: TNNTPAuthenticateResponse;
    FOnBanner: TNNTPBanner;
    FOnSelectGroup: TNNTPSelectGroup;
    FOnNextArticle: TNNTPNextArticle;
    FOnlastArticle: TNNTPlastArticle;
    FOnArticleStatus: TNNTPArticleStatus;
    function Get_Timeout(event: Smallint): Integer; stdcall;
    procedure Set_Timeout(event: Smallint; Value: Integer); stdcall;
    procedure Set_EnableTimer(event: Smallint; Value: TOleBool); stdcall;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure Cancel; stdcall;
    procedure Connect(const RemoteHost, RemotePort: Variant); stdcall;
    procedure SendDoc(const URL, Headers, InputData, InputFile, OutputFile:
        Variant); stdcall;
    procedure GetDoc(const URL, Headers, OutputFile: Variant); stdcall;
    procedure GetAdministrationFile(const LastUpdate: Variant); stdcall;
    procedure SelectGroup(const groupName: string); stdcall;
    procedure SetNextArticle; stdcall;
    procedure SetLastArticle; stdcall;
    procedure GetArticleNumbers(const groupName: Variant); stdcall;
    procedure GetArticleHeaders(const header: string; const firstArticle,
        lastArticle: Variant); stdcall;
    procedure GetArticleByArticleNumber(const articleNumber: Variant);
        stdcall;
    procedure GetArticleByMessageID(const messageID: string); stdcall;
```

```
    procedure GetHeaderByArticleNumber(const articleNumber: Variant);
        stdcall;
    procedure GetHeaderByMessageID(const messageID: string); stdcall;
    procedure GetBodyByArticleNumber(const articleNumber: Variant);
        stdcall;
    procedure GetBodyByMessageID(const messageID: string); stdcall;
    procedure GetStatByArticleNumber(const articleNumber: Variant);
        stdcall;
    procedure GetOverviewFormat; stdcall;
    procedure GetOverview(const firstArticle, lastArticle: Variant);
        stdcall;
    procedure ListGroups; stdcall;
    procedure ListGroupDescriptions; stdcall;
    procedure ListNewGroups(const LastUpdate: Variant); stdcall;
    procedure Quit; stdcall;
    property State: Smallint index 503 read GetSmallintProp;
    property ProtocolState: Smallint index 504 read GetSmallintProp;
    property ReplyString: string index 505 read GetStringProp;
    property ReplyCode: Integer index 506 read GetIntegerProp;
    property Errors: Variant index 508 read GetVariantProp;
    property Busy: TOleBool index 509 read GetOleBoolProp;
    property StateString: string index 511 read GetStringProp;
    property ProtocolStateString: string index 512 read GetStringProp;
    property DocInput: Variant index 1002 read GetVariantProp;
    property DocOutput: Variant index 1003 read GetVariantProp;
    property ArticleNumbersSupported: TOleBool index 1 read GetOleBoolProp;
    property OverviewSupported: TOleBool index 2 read GetOleBoolProp;
    property PostingAllowed: TOleBool index 3 read GetOleBoolProp;
    property Timeout[event: Smallint]: Integer read Get_Timeout write
        Set_Timeout;
    property EnableTimer[event: Smallint]: TOleBool write Set_EnableTimer;
  published
    property RemoteHost: string index 0 read GetStringProp write
        SetStringProp stored False;
    property RemotePort: Integer index 502 read GetIntegerProp write
        SetIntegerProp stored False;
    property NotificationMode: Smallint index 510 read GetSmallintProp
        write SetSmallintProp stored False;
    property Logging: TOleBool index 514 read GetOleBoolProp write
        SetOleBoolProp stored False;
    property URL: string index 1001 read GetStringProp write SetStringProp
        stored False;
    property LastUpdate: TOleDate index 4 read GetOleDateProp write
        SetOleDateProp stored False;
    property OnError: TNNTPError read FOnError write FOnError;
    property OnTimeout: TNNTPTimeout read FOnTimeout write FOnTimeout;
    property OnCancel: TNotifyEvent read FOnCancel write FOnCancel;
    property OnStateChanged: TNNTPStateChanged read FOnStateChanged write
        FOnStateChanged;
    property OnProtocolStateChanged: TNNTPProtocolStateChanged read
        FOnProtocolStateChanged write FOnProtocolStateChanged;
    property OnBusy: TNNTPBusy read FOnBusy write FOnBusy;
    property OnLog: TNotifyEvent read FOnLog write FOnLog;
    property OnDocInput: TNNTPDocInput read FOnDocInput write FOnDocInput;
    property OnDocOutput: TNNTPDocOutput read FOnDocOutput write
        FOnDocOutput;
    property OnAuthenticateRequest: TNNTPAuthenticateRequest read
        FOnAuthenticateRequest write FOnAuthenticateRequest;
    property OnAuthenticateResponse: TNNTPAuthenticateResponse read
        FOnAuthenticateResponse write FOnAuthenticateResponse;
    property OnBanner: TNNTPBanner read FOnBanner write FOnBanner;
```

```
    property OnSelectGroup: TNNTPSelectGroup read FOnSelectGroup write
        FOnSelectGroup;
    property OnNextArticle: TNNTPNextArticle read FOnNextArticle write
        FOnNextArticle;
    property OnlastArticle: TNNTPlastArticle read FOnlastArticle write
        FOnlastArticle;
    property OnArticleStatus: TNNTPArticleStatus read FOnArticleStatus
        write FOnArticleStatus;
  end;
```

Example:

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, OleCtrls, StdCtrls, ComCtrls, ExtCtrls, Buttons, OLE2, ISP;

const
  efListGroups = 0;
  efGetArticleHeaders = 1;
  efGetArticleNumbers = 2;
  efGetArticle = 3;

type
  TNewsForm = class(TForm)
    NNTP1: TNNTP;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    N1: TMenuItem;
    FileDisconnectItem: TMenuItem;
    FileConnectItem: TMenuItem;
    Panel1: TPanel;
    Bevel1: TBevel;
    StatusBar: TStatusBar;
    SmallImages: TImageList;
    Panel2: TPanel;
    NewsGroups: TTreeView;
    Bevel2: TBevel;
    Panel3: TPanel;
    Memo1: TMemo;
    Panel5: TPanel;
    Panel4: TPanel;
    ConnectBtn: TSpeedButton;
    RefreshBtn: TSpeedButton;
    Bevel3: TBevel;
    MsgHeaders: TListBox;
    Label1: TLabel;
    Label2: TLabel;
    procedure FileConnectItemClick(Sender: TObject);
    procedure NNTP1AuthenticateRequest(Sender: TObject; var UserId,
      Password: string);
    procedure NNTP1ProtocolStateChanged(Sender: TObject;
      ProtocolState: Smallint);
    procedure NNTP1DocOutput(Sender: TObject; const DocOutput: Variant);
    procedure NNTP1StateChanged(Sender: TObject; State: Smallint);
    procedure Exit1Click(Sender: TObject);
    procedure NNTP1Error(Sender: TObject; Number: Smallint;
      var Description: string; Scode: Integer; const Source,
```

```
      HelpFile: string; HelpContext: Integer; var CancelDisplay: Wordbool);
    procedure NNTP1Banner(Sender: TObject; const Banner: string);
    procedure MsgHeadersDblClick(Sender: TObject);
    procedure NNTP1SelectGroup(Sender: TObject; const groupName: string;
      firstMessage, lastMessage, msgCount: Integer);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure NewsGroupsChange(Sender: TObject; Node: TTreeNode);
    procedure RefreshBtnClick(Sender: TObject);
    procedure FileDisconnectItemClick(Sender: TObject);
  private
    EventFlag: Integer;
    function NodePath(Node: TTreeNode): String;
  public
    Data: String;
  end;

var
  NewsForm: TNewsForm;
  Remainder: String;
  Nodes: TStringList;
  CurrentGroup: String;
  GroupCount: Integer;

implementation

uses Connect;
{$R *.DFM}

{ TParser }

type

  TToken = (etEnd, etSymbol, etName, etLiteral);

  TParser = class
  private
    FFlags: Integer;
    FText: string;
    FSourcePtr: PChar;
    FSourceLine: Integer;
    FTokenPtr: PChar;
    FTokenString: string;
    FToken: TToken;
    procedure SkipBlanks;
    procedure NextToken;
    function TokenName: string;
    function TokenSymbolIs(const S: string): Boolean;
  public
    constructor Create(const Text: string; Groups: Boolean);
  end;

const
  sfAllowSpaces = 1;

constructor TParser.Create(const Text: string; Groups: Boolean);
begin
  FText := Text;
  FSourceLine := 1;
  FSourcePtr := PChar(Text);
  if Groups then
    FFlags := sfAllowSpaces
```

```
  else
    FFlags := 0;
  NextToken;
end;

procedure TParser.SkipBlanks;
begin
  while True do
  begin
    case FSourcePtr^ of
      #0:
        begin
          if FSourcePtr^ = #0 then Exit;
          Continue;
        end;
      #10:
        Inc(FSourceLine);
      #33..#255:
        Exit;
    end;
    Inc(FSourcePtr);
  end;
end;

procedure TParser.NextToken;
var
  P, TokenStart: PChar;
  L: Integer;
  StrBuf: array[0..255] of Char;
begin
  SkipBlanks;
  FTokenString := '';
  P := FSourcePtr;
  while (P^ <> #0) and (P^ <= ' ') do Inc(P);
  FTokenPtr := P;
  case P^ of
    '0'..'9':
      begin
        TokenStart := P;
        Inc(P);
        while P^ in ['0'..'9'] do Inc(P);
        SetString(FTokenString, TokenStart, P - TokenStart);
        FToken := etLiteral;
      end;
    #13: Inc(FSourceLine);
    #0:
      FToken := etEnd;
  else
    begin
      TokenStart := P;
      Inc(P);
      if FFlags = sfAllowSpaces then
        while not (P^ in [#0, #13, ' ']) do Inc(P)
      else
        while not (P^ in [#0, #13]) do Inc(P);
      SetString(FTokenString, TokenStart, P - TokenStart);
      FToken := etSymbol;
    end;
  end;
  FSourcePtr := P;
end;
```

```
function TParser.TokenName: string;
begin
  if FSourcePtr = FTokenPtr then Result := LoadStr(0) else
  begin
    SetString(Result, FTokenPtr, FSourcePtr - FTokenPtr);
    Result := '''' + Result + '''';
  end;
end;

function TParser.TokenSymbolIs(const S: string): Boolean;
begin
  Result := (FToken = etSymbol) and (CompareText(FTokenString, S) = 0);
end;

function FirstItem(var ItemList: ShortString): ShortString;
var
  P: Integer;
begin
  P := Pos('.', ItemList);
  if P = 0 then
  begin
    Result := ItemList;
    P := Length(ItemList);
  end
  else
    Result := Copy(ItemList, 1, P - 1);
  Delete(ItemList, 1, P);
end;

procedure AddItem(GroupName: ShortString);
var
  Name, NewGroup: String;
  LeafText: String;
  Index, i: Integer;
  Groups: Integer;
  Item: ShortString;
  TheNodes: TStringList;
begin
  Groups := 1;
  for i := 0 to Length(GroupName) do
    if GroupName[i] = '.' then
      Inc(Groups);
  TheNodes := Nodes;
  for i := 0 to Groups - 1 do
  begin
    Item := FirstItem(GroupName);
    Index := TheNodes.IndexOf(Item);
    if Index = -1 then
    begin
      Index := TheNodes.AddObject(Item, TStringList.Create);
      TheNodes := TStringList(TheNodes.Objects[Index]);
      TheNodes.Sorted := True;
    end
    else
      TheNodes := TStringList(TheNodes.Objects[Index]);
  end;
  Inc(GroupCount);
end;
```

```
procedure ParseGroups(Data: String);
var
  Parser: TParser;
  OldSrcLine: Integer;
begin
  Parser := TParser.Create(Data, True);
  OldSrcLine := 0;
  while Parser.FToken <> etEnd do
  begin
    if Parser.FSourceLine <> OldSrcLine then
    begin
      AddItem(Parser.FTokenString);
      OldSrcLine := Parser.FSourceLine;
    end;
    Parser.NextToken;
  end;
end;

procedure ParseHeaders(Data: String);
var
  Parser: TParser;
  MsgNo: LongInt;
  Header: String;
  OldSrcLine: Integer;
begin
  Parser := TParser.Create(Data, False);
  while Parser.FToken <> etEnd do
  begin
    MsgNo := StrToInt(Parser.FTokenString);
    OldSrcLine := Parser.FSourceLine;
    Parser.NextToken;
    Header := '';
    while (OldSrcLine = Parser.FSourceLine) do
    begin
      Header := Header + ' ' + Parser.FTokenString;
      Parser.NextToken;
      if Parser.FToken = etEnd then
        Break;
    end;
    NewsForm.MsgHeaders.Items.AddObject(Header, Pointer(MsgNo));
  end;
end;

procedure DestroyList(AList: TStringList);
var
  i: Integer;
begin
  for i := 0 to AList.Count - 1 do
    if AList.Objects[i] <> nil then
      DestroyList(TStringList(AList.Objects[i]));
  AList.Free;
end;

procedure BuildTree(Parent: TTreeNode; List: TStrings);
var
  i: Integer;
  Node: TTreeNode;
begin
  for i := 0 to List.Count - 1 do
    if List.Objects[i] <> nil then
    begin
```

```
      Node := NewsForm.NewsGroups.Items.AddChild(Parent, List[i]);
      Node.ImageIndex := 0;
      Node.SelectedIndex := 1;
      BuildTree(Node, TStrings(List.Objects[i]));
    end
    else
      NewsForm.NewsGroups.Items.AddChild(Parent, List[i]);
end;

function TNewsForm.NodePath(Node: TTreeNode): String;
begin
  if Node.Parent = nil then
    Result := Node.Text
  else
    Result := NodePath(Node.Parent) + '.' + Node.Text;
end;

procedure TNewsForm.FileConnectItemClick(Sender: TObject);
begin
  ConnectDlg := TConnectDlg.Create(Self);
  try
    if ConnectDlg.ShowModal = mrOk then
      with NNTP1 do
        Connect(ConnectDlg.ServerEdit.Text, RemotePort);
  finally
    ConnectDlg.Free;
  end;
end;

procedure TNewsForm.NNTP1AuthenticateRequest(Sender: TObject; var UserId,
  Password: string);
begin
  UserID := '';
  Password := '';
end;

procedure TNewsForm.NNTP1ProtocolStateChanged(Sender: TObject;
  ProtocolState: Smallint);
begin
  case ProtocolState of
    nntpBase: ;
    nntpTransaction:
      begin
        EventFlag := efListGroups;
        Nodes := TStringList.Create;
        Nodes.Sorted := True;
        NNTP1.ListGroups;
      end;
  end;
end;

procedure TNewsForm.NNTP1DocOutput(Sender: TObject; const DocOutput:
Variant);
var
  S: String;
  i: integer;
  MsgNo, Header: String;
begin
  Statusbar.Panels[2].Text := Format('Bytes: %s',
[DocOutput.BytesTransferred]);
  case DocOutput.State of
```

```
    icDocBegin:
      begin
        if EventFlag = efListGroups then
          Memo1.Lines.Add('Retrieving news groups...');
        Data := '';
        GroupCount := 0;
      end;
    icDocData:
      begin
        DocOutput.GetData(S, VT_BSTR);
        Data := Data + S;
        if EventFlag = efGetArticle then
          Memo1.Lines.Add(S);
      end;
    icDocEnd:
      begin
        case EventFlag of
          efListGroups:
            begin
              ParseGroups(Data);
              Memo1.Lines.Add('Done.'#13#10'Building news group tree...');
              NewsGroups.Items.BeginUpdate;
              try
                BuildTree(nil, Nodes);
                DestroyList(Nodes);
                Statusbar.Panels[1].Text := Format('%d Groups',
[GroupCount]);
              finally
                NewsGroups.Items.EndUpdate;
                Memo1.Lines.Add('Done.');
              end;
            end;
          efGetArticleHeaders: ParseHeaders(Data);
          efGetArticle:
            begin
              Memo1.SelStart := 0;
              SendMessage(Memo1.Handle, EM_ScrollCaret, 0, 0);
            end;
        end;
        SetLength(Data, 0);
      end;
  end;
  Refresh;
end;

procedure TNewsForm.NNTP1StateChanged(Sender: TObject; State: Smallint);
begin
  with Memo1.Lines do
    case NNTP1.State of
      prcConnecting   : Add('Connecting');
      prcResolvingHost: Add('Resolving Host: ' + NNTP1.RemoteHost);
      prcHostResolved : Add('Host resolved');
      prcConnected    :
        begin
          Add('Connected to: ' + NNTP1.RemoteHost);
          Statusbar.Panels[0].Text := 'Connected to: ' + NNTP1.RemoteHost;
          ConnectBtn.Enabled := False;
          RefreshBtn.Enabled := True;
        end;
      prcDisconnecting: Text := NNTP1.ReplyString;
      prcDisconnected :
```

```
        begin
          Statusbar.Panels[0].Text := 'Disconnected';
          Caption := 'News Reader';
          Label1.Caption := '';
          ConnectBtn.Enabled := True;
          RefreshBtn.Enabled := False;
        end;
    end;
end;

procedure TNewsForm.Exit1Click(Sender: TObject);
begin
  if NNTP1.State <> prcDisconnected then
  begin
    if NNTP1.Busy then NNTP1.Cancel;
    NNTP1.Quit;
    while NNTP1.State <> prcDisconnected do
      Application.ProcessMessages;
  end;
  Close;
end;

procedure TNewsForm.NNTP1Error(Sender: TObject; Number: Smallint;
  var Description: string; Scode: Integer; const Source, HelpFile: string;
  HelpContext: Integer; var CancelDisplay: Wordbool);
begin
//  MessageDlg(Description, mtError, [mbOk], 0);
end;

procedure TNewsForm.NNTP1Banner(Sender: TObject; const Banner: string);
begin
  Memo1.Lines.Add(Banner);
end;

procedure TNewsForm.NNTP1SelectGroup(Sender: TObject; const groupName:
string;
  firstMessage, lastMessage, msgCount: Integer);
begin
  EventFlag := efGetArticleHeaders;
  Statusbar.Panels[1].Text := Format('%d Article(s)',[msgCount]);
  NNTP1.GetArticleHeaders('subject', FirstMessage, lastMessage);
end;

procedure TNewsForm.MsgHeadersDblClick(Sender: TObject);
var
  Article: Integer;
begin
  if NNTP1.Busy then exit;
  EventFlag := efGetArticle;
  Memo1.Clear;
  if MsgHeaders.ItemIndex = -1 then exit;
  Caption := 'News Reader: ' + MsgHeaders.Items[MsgHeaders.ItemIndex];
  Article := Integer(MsgHeaders.Items.Objects[MsgHeaders.ItemIndex]);
  NNTP1.GetArticlebyArticleNumber(Article);
end;

procedure TNewsForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if NNTP1.State <> prcDisconnected then
  begin
    if NNTP1.Busy then NNTP1.Cancel;
```

```
    NNTP1.Quit;
    while NNTP1.State <> prcDisconnected do
      Application.ProcessMessages;
  end;
end;

procedure TNewsForm.NewsGroupsChange(Sender: TObject; Node: TTreeNode);
var
  NP: String;
begin
  if (NNTP1.State = prcConnected) and not NNTP1.Busy then
    with MsgHeaders do
    begin
      Items.BeginUpdate;
      try
        Items.Clear;
        Memo1.Lines.Clear;
        NP := NodePath(NewsGroups.Selected);
        Statusbar.Panels[2].Text := 'Bytes: 0';
        Statusbar.Panels[1].Text := '0 Article(s)';
        if NNTP1.Busy then
          NNTP1.Cancel;
        NNTP1.SelectGroup(NP);
        Label1.Caption := 'Contents of ''' + NP + '''';
      finally
        Items.EndUpdate;
      end;
    end;
end;

procedure TNewsForm.RefreshBtnClick(Sender: TObject);
begin
  if NewsGroups.Selected <> nil then
    NewsGroupsChange(nil, NewsGroups.Selected);
end;

procedure TNewsForm.FileDisconnectItemClick(Sender: TObject);
begin
  if NNTP1.Busy then NNTP1.Cancel;
  NNTP1.Quit;
  while NNTP1.Busy do
    Application.ProcessMessages;
  with NewsGroups.Items do
  begin
    BeginUpdate;
    Clear;
    EndUpdate;
  end;
  MsgHeaders.Items.Clear;
  Memo1.Lines.Clear;
end;

end.
```

POP - Post Office Protocol

The POP Client Control implements the POP3 Protocol Client as specified by RFC 1081, Post Office Protocol. It provides access to Internet mail servers using the POP3 protocol. It can be used by Internet mail developers or system integrators. The major advantage of this control is its ability to retrieve mail from UNIX or other servers supporting POP3 protocol.

The POP Client Control has the ability to connect to a server, send authentication information (user and password) to the server, retrieve user mailbox information such as the number of messages waiting to be retrieved, retrieve messages from the server, and delete messages from the server.

Properties:

Busy	Indicates a command is in progress.
DocOutput	Object describing output information for the document being transferred. The DocOutput object provides a more powerful interface than the basic capabilities of the GetDoc method.
EnableTimer	Boolean property to enable timer for the specified event. Value is specified in the TimeOut property.
Errors	A collection of errors that can be accessed for details about the last error that occurred. This collection should be used within an Error event if information passed through the Error event is not sufficient.
MessageCount	This property specifies the number of messages in the mailbox. It is established after authentication has been successfully performed. Before that it is invalid.
NotificationMode	Determines when notification is issued for incoming data. Notification can also be suspended. 0 = COMPLETE: notification is provided when there is a complete response. 1 = CONTINUOUS: an event is repeatedly activated when new data arrives from the connection.
Password	Password of current user on the FTP Server.
ProtocolState	This property specifies the current state of the protocol. prcBase = 0 Base state before connection to server is established. prcAuthorization= 1 Authorization is being performed. prcTransaction = 2 Authorization had been performed successfully, the client has successfully identified itself to the POP3 server and the POP3 server has locked and burst the appropriate maildrop. prcUpdate = 3 When Quit command is issued

from transaction state.

ProtocolStateString	String representation of ProtocolState.
RemoteHost	The remote machine to connect to if the remoteHost parameter in the Connect method is missing. You can either provide a host name or an IP address string in dotted format. For example, 127.0.0.1.
RemotePort	The remote port number to which to connect.
ReplyCode	The value of the reply code is a protocol specific number that determines the result of the last request, as returned in the ReplyString property. See RFC 1081 for a list of valid reply codes.
ReplyString	Line returned to the client as a result of a request.
State	This property specifies the connection state of the control. prcConnecting = 1 - Connecting. Connect has been requested, waiting for connect acknowledge. prcResolvingHost = 2 - Resolving Host. Occurs when RemoteHost is in name format rather than dot-delimited IP format. prcHostResolved = 3 - Resolved the host. Occurs only if ResolvingHost state has been entered previously. prcConnected = 4 - Connection established. prcDisconnecting = 5 - Connection closed. Disconnect has been initiated. prcDisconnected = 6 - Initial state when protocol object is instantiated, before Connect has been initiated, after a Connect attempt failed or after Disconnect performed.
StateString	A string representation of State.
TimeOut	Timeout value for the specified event.
TopLines	Designates the number of lines to be retrieved in a top request.
TopSupported	This property indicates "Top is supported". It can be queried after a connection to the server has been established. It is set to TRUE if the particular server supports the TOP command.
UserId	User identification name for the client on the server.
URL	URL string identifying the current document being transferred. The URL format for this control is: POP://user:password@host:port /message number In the POP Control, the URL property may identify a message being retrieved from a remote server.

Methods:

Authenticate	Authenticates the user based on the parameters passed. If no parameters are passed, the UserId and Password properties are used. If neither the UserId nor the Password are entered, the control uses the URL. When authentication process is terminated, the Authenticate event is activated.
Cancel	Cancels a pending request.
Connect	Initiates a Connect request. The control calls the StateChanged event if a connection is established.
Delete	Initiates a Delete request. If successful, a Delete event is activated, otherwise an Error event is activated.
GetDoc	A DocOutput related method that requests retrieval of a document identified by a URL. The GetDoc method in POP gets a message from the server.
Last	Initiates a LAST request. If successful, the LAST event is activated. This request is used to find the highest message number accessed by the client.
MessageSize	Initiates a request to retrieve the message size. If successful, a MessageSize event is activated, otherwise the Error event is activated.
NOOP	Initiates a NOOP request. This is used to test the connection.
Quit	Initiates a Quit request. If unsuccessful, the Error event is activated.
RefreshMessageCount	This method will refresh the number of undeleted messages from your current maildrop. When the request is completed, the RefreshMessageCount event is activated, indicating the current number of undeleted messages. This method is only available if you are already connected and authenticated.
Reset	Initiate a RSET request. Any messages marked as deleted will be unmarked. If successful, a corresponding Reset event is activated, otherwise an Error event is activated.
RetrieveMessage	Initiates a RetreiveMessage request for the message specified in msgNumber. DocOutput event can be used to retrieve the data.
TopMessage	Initiates a Top of Message request for the message specified in msgNumber. TopMessage is used in conjunction with the TopLines property. If TopLines is 0, then only header information will be retrieved.

Events:

OnBusy — This event is activated when a command is in progress or when a command has completed.

OnCancel — This event is activated after a cancellation request has been completed and satisfied. After this event the object's state changes to idle.

OnDelete — This event is activated after the successful completion of a Delete request.

OnDocOutput — A DocOutput related event indicating that output data has been transferred. The DocOutput event can be used in its basic form for notification during transfer.

OnError — This event is activated when an error occurs in background processing.

POP Error Codes : The following error codes apply only to the POP ActiveX Control.

Error Number	Error Message
2450	RetrieveMessage Command Failed. Unable to retrieve message.
2451	Delete Command Failed. Unable to delete message.
2452	Reset Command Failed. Unable to unmark deleted message(s).
2453	Last Command Failed. Unable to find the highest message number accessed by client.
2454	RefreshMessageCount Command Failed. Unable to ascertain the number of messages marked as deleted.
2455	Noop Command Failed. Unable to test the connection.
2456	Quit Command Failed. Error while quitting.
2457	TopMessage Command Failed. Unable to retrieve the TopLines of the message.

OnLast — This event is activated after the successful completion of a Last request. It indicates the number of the last message accessed by the client.

OnMessageSize — This event is activated after successful completion of a MessageSize request.

OnNoop — This event is activated after the NOOP method is called. It requests an OK reply from the server.

OnProtocolStateChanged — This event is activated whenever the protocol state changes.

OnRefreshMessageCount — This event is activated after a successful completion of RefreshMessageCount request. The number of undeleted messages from the current maildrop is returned. (A maildrop contains the

	messages that can be retrieved/deleted in the current state.)
OnReset	This event is activated after the successful completion of a Reset request.
OnStateChanged	This event is activated whenever the state of the transport state changes.
OnTimeout	This event is activated when the timer for the specified event expires. See Timeout property for pre-defined events.

As Declared in ISP.PAS:

```
{ TPOP }

TPOPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
TPOPTimeout = procedure(Sender: TObject; event: Smallint; var Continue:
TOleBool) of object;
TPOPStateChanged = procedure(Sender: TObject; State: Smallint) of object;
TPOPProtocolStateChanged = procedure(Sender: TObject; ProtocolState:
Smallint) of object;
TPOPBusy = procedure(Sender: TObject; isBusy: TOleBool) of object;
TPOPDocOutput = procedure(Sender: TObject; const DocOutput: Variant) of
object;
TPOPMessageSize = procedure(Sender: TObject; msgSize: Integer) of object;
TPOPLast = procedure(Sender: TObject; Number: Integer) of object;
TPOPRefreshMessageCount = procedure(Sender: TObject; Number: Integer) of
object;

TPOP = class(TOleControl)
  private
    FOnError: TPOPError;
    FOnTimeout: TPOPTimeout;
    FOnCancel: TNotifyEvent;
    FOnStateChanged: TPOPStateChanged;
    FOnProtocolStateChanged: TPOPProtocolStateChanged;
    FOnBusy: TPOPBusy;
    FOnLog: TNotifyEvent;
    FOnDocOutput: TPOPDocOutput;
    FOnMessageSize: TPOPMessageSize;
    FOnDelete: TNotifyEvent;
    FOnReset: TNotifyEvent;
    FOnLast: TPOPLast;
    FOnNoop: TNotifyEvent;
    FOnRefreshMessageCount: TPOPRefreshMessageCount;
    function Get_Timeout(event: Smallint): Integer; stdcall;
    procedure Set_Timeout(event: Smallint; Value: Integer); stdcall;
    procedure Set_EnableTimer(event: Smallint; Value: TOleBool); stdcall;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure Cancel; stdcall;
    procedure Connect(const RemoteHost, RemotePort: Variant); stdcall;
    procedure Authenticate(const UserId, Password: Variant); stdcall;
    procedure GetDoc(const URL, Headers, OutputFile: Variant); stdcall;
    procedure MessageSize(MessageNumber: Smallint); stdcall;
    procedure RetrieveMessage(MessageNumber: Smallint); stdcall;
```

```
  procedure Delete(MessageNumber: Smallint); stdcall;
  procedure Reset; stdcall;
  procedure Last; stdcall;
  procedure Noop; stdcall;
  procedure TopMessage(MessageNumber: Smallint); stdcall;
  procedure Quit; stdcall;
  procedure RefreshMessageCount; stdcall;
  property State: Smallint index 503 read GetSmallintProp;
  property ProtocolState: Smallint index 504 read GetSmallintProp;
  property ReplyString: string index 505 read GetStringProp;
  property ReplyCode: Integer index 506 read GetIntegerProp;
  property Errors: Variant index 508 read GetVariantProp;
  property Busy: TOleBool index 509 read GetOleBoolProp;
  property StateString: string index 511 read GetStringProp;
  property ProtocolStateString: string index 512 read GetStringProp;
  property DocOutput: Variant index 1003 read GetVariantProp;
  property TopSupported: TOleBool index 2452 read GetOleBoolProp;
  property MessageCount: Smallint index 2454 read GetSmallintProp;
  property Timeout[event: Smallint]: Integer read Get_Timeout write
      Set_Timeout;
  property EnableTimer[event: Smallint]: TOleBool write Set_EnableTimer;
published
  property RemoteHost: string index 0 read GetStringProp write
      SetStringProp stored False;
  property RemotePort: Integer index 502 read GetIntegerProp write
      SetIntegerProp stored False;
  property NotificationMode: Smallint index 510 read GetSmallintProp
      write SetSmallintProp stored False;
  property Logging: TOleBool index 514 read GetOleBoolProp write
      SetOleBoolProp stored False;
  property UserId: string index 601 read GetStringProp write
      SetStringProp stored False;
  property Password: string index 602 read GetStringProp write
      SetStringProp stored False;
  property URL: string index 1001 read GetStringProp write SetStringProp
      stored False;
  property TopLines: Integer index 2453 read GetIntegerProp write
      SetIntegerProp stored False;
  property OnError: TPOPError read FOnError write FOnError;
  property OnTimeout: TPOPTimeout read FOnTimeout write FOnTimeout;
  property OnCancel: TNotifyEvent read FOnCancel write FOnCancel;
  property OnStateChanged: TPOPStateChanged read FOnStateChanged write
      FOnStateChanged;
  property OnProtocolStateChanged: TPOPProtocolStateChanged read
      FOnProtocolStateChanged write FOnProtocolStateChanged;
  property OnBusy: TPOPBusy read FOnBusy write FOnBusy;
  property OnLog: TNotifyEvent read FOnLog write FOnLog;
  property OnDocOutput: TPOPDocOutput read FOnDocOutput write
      FOnDocOutput;
  property OnMessageSize: TPOPMessageSize read FOnMessageSize write
      FOnMessageSize;
  property OnDelete: TNotifyEvent read FOnDelete write FOnDelete;
  property OnReset: TNotifyEvent read FOnReset write FOnReset;
  property OnLast: TPOPLast read FOnLast write FOnLast;
  property OnNoop: TNotifyEvent read FOnNoop write FOnNoop;
  property OnRefreshMessageCount: TPOPRefreshMessageCount read
      FOnRefreshMessageCount write FOnRefreshMessageCount;
end;
```

Example:

```
unit Main;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls, OleCtrls,
  ISP;

type

  TMainForm = class(TForm)
    OpenDialog: TOpenDialog;
    SMTP1: TSMTP;
    POP1: TPOP;
    PageControl1: TPageControl;
    SendPage: TTabSheet;
    RecvPage: TTabSheet;
    ConPage: TTabSheet;
    Panel1: TPanel;
    Label1: TLabel;
    Label3: TLabel;
    Label2: TLabel;
    eTo: TEdit;
    eCC: TEdit;
    eSubject: TEdit;
    SendBtn: TButton;
    ClearBtn: TButton;
    reMessageText: TRichEdit;
    SMTPStatus: TStatusBar;
    Panel3: TPanel;
    mReadMessage: TMemo;
    POPStatus: TStatusBar;
    cbSendFile: TCheckBox;
    GroupBox1: TGroupBox;
    ePOPServer: TEdit;
    Label6: TLabel;
    Label5: TLabel;
    eUserName: TEdit;
    ePassword: TEdit;
    Label4: TLabel;
    GroupBox2: TGroupBox;
    Label7: TLabel;
    eSMTPServer: TEdit;
    SMTPConnectBtn: TButton;
    POPConnectBtn: TButton;
    eHomeAddr: TEdit;
    Label8: TLabel;
    Panel2: TPanel;
    Label9: TLabel;
    lMessageCount: TLabel;
    Label10: TLabel;
    eCurMessage: TEdit;
    udCurMessage: TUpDown;
    ConnectStatus: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure POP1StateChanged(Sender: TObject; State: Smallint);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure SMTP1StateChanged(Sender: TObject; State: Smallint);
    procedure SMTP1DocInput(Sender: TObject; const DocInput: Variant);
    procedure SMTP1Error(Sender: TObject; Number: Smallint;
```

```
      var Description: string; Scode: Integer; const Source,
      HelpFile: string; HelpContext: Integer; var CancelDisplay: Wordbool);
    procedure FormResize(Sender: TObject);
    procedure ClearBtnClick(Sender: TObject);
    procedure SMTP1Verify(Sender: TObject);
    procedure SendBtnClick(Sender: TObject);
    procedure POP1ProtocolStateChanged(Sender: TObject;
      ProtocolState: Smallint);
    procedure POP1Error(Sender: TObject; Number: Smallint;
      var Description: string; Scode: Integer; const Source,
      HelpFile: string; HelpContext: Integer; var CancelDisplay: Wordbool);
    procedure SMTPConnectBtnClick(Sender: TObject);
    procedure POPConnectBtnClick(Sender: TObject);
    procedure eSMTPServerChange(Sender: TObject);
    procedure ePOPServerChange(Sender: TObject);
    procedure cbSendFileClick(Sender: TObject);
    procedure POP1DocOutput(Sender: TObject; const DocOutput: Variant);
    procedure udCurMessageClick(Sender: TObject; Button: TUDBtnType);
    procedure POP1RefreshMessageCount(Sender: TObject; Number: Integer);
  private
    RecvVerified,
    SMTPError,
    POPError: Boolean;
    FMessageCount: Integer;
    procedure SendFile(Filename: string);
    procedure SendMessage;
    procedure CreateHeaders;
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

uses OLEAuto;

const
  icDocBegin = 1;
  icDocHeaders = 2;
  icDocData = 3;
  icDocEnd = 5;

{When calling a component method which maps onto an OLE call, NoParam
substitutes for an optional parameter. As an alternative to calling the
component method, you may access the component's OLEObject directly -
i.e., Component.OLEObject.MethodName(,Foo,,Bar)}
function NoParam: Variant;
begin
  TVarData(Result).VType := varError;
  TVarData(Result).VError := DISP_E_PARAMNOTFOUND;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  SMTPError := False;
  POPError := False;
  FMessageCount := 0;
end;
```

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if POP1.State = prcConnected then POP1.Quit;
  if SMTP1.State = prcConnected then SMTP1.Quit;
end;

procedure TMainForm.FormResize(Sender: TObject);
begin
  SendBtn.Left := ClientWidth - SendBtn.Width - 10;
  ClearBtn.Left := ClientWidth - ClearBtn.Width - 10;
  cbSendFile.Left := ClientWidth - cbSendFile.Width - 10;
  eTo.Width := SendBtn.Left - eTo.Left - 10;
  eCC.Width := SendBtn.Left - eCC.Left - 10;
  eSubject.Width := SendBtn.Left - eSubject.Left - 10;
end;

procedure TMainForm.ClearBtnClick(Sender: TObject);
begin
  eTo.Text := '';
  eCC.Text := '';
  eSubject.Text := '';
  OpenDialog.Filename := '';
  reMessageText.Lines.Clear;
end;

procedure TMainForm.eSMTPServerChange(Sender: TObject);
begin
  SMTPConnectBtn.Enabled:=(eSMTPServer.Text<>'') and (eHomeAddr.Text<>'');
end;

procedure TMainForm.ePOPServerChange(Sender: TObject);
begin
  POPConnectBtn.Enabled:=(ePOPServer.Text <> '') and (eUsername.Text <> '')
    and (ePassword.Text <> '');
end;

procedure TMainForm.cbSendFileClick(Sender: TObject);
begin
  if cbSendFile.Checked then
  begin
    if OpenDialog.Execute then
      cbSendFile.Caption := cbSendFile.Caption + ': '+OpenDialog.Filename
    else
      cbSendFile.Checked := False;
  end
  else
    cbSendFile.Caption := '&Attach Text File';
end;

{Clear and repopulate MIME headers, using the component's DocInput
property. A separate DocInput OLE object could also be used. See RFC 1521 /
RFC 1522 for complete information on MIME types.}
procedure TMainForm.CreateHeaders;
begin
  with SMTP1 do
  begin
    DocInput.Headers.Clear;
    DocInput.Headers.Add('To', eTo.Text);
    DocInput.Headers.Add('From', eHomeAddr.Text);
    DocInput.Headers.Add('CC', eCC.Text);
    DocInput.Headers.Add('Subject', eSubject.Text);
```

```
    DocInput.Headers.Add('Message-Id', Format('%s_%s_%s',
        [Application.Title, DateTimeToStr(Now), eHomeAddr.Text]));
    DocInput.Headers.Add('Content-Type', 'TEXT/PLAIN charset=US-ASCII');
  end;
end;

{Send a simple mail message}
procedure TMainForm.SendMessage;
begin
  CreateHeaders;
  with SMTP1 do
    SendDoc(NoParam, DocInput.Headers, reMessageText.Text, '', '');
end;

{Send a disk file. Leave SendDoc's InputData parameter blank and specify a
filename for InputFile to send the contents of a disk file. You can use the
DocInput event and GetData methods to do custom encoding (Base64, UUEncode,
etc.) }
procedure TMainForm.SendFile(Filename: string);
begin
  CreateHeaders;
  with SMTP1 do
  begin
    DocInput.Filename := FileName;
    SendDoc(NoParam, DocInput.Headers, NoParam, DocInput.FileName, '');
  end;
end;

{Set global flag indicating recipients are addressable (this only ensures
that the address is in the correct format, not that it exists and is
deliverable), then send the text part of the message}
procedure TMainForm.SMTP1Verify(Sender: TObject);
begin
  SendMessage;
  RecvVerified := True;
end;

{Verify addressees, send text message in the Verify event, and if an
attachment is specified, send it}
procedure TMainForm.SendBtnClick(Sender: TObject);
var
  Addressees: string;
begin
  if SMTP1.State = prcConnected then
  begin
    RecvVerified := False;
    SMTPError := False;
    Addressees := eTo.Text;

    if eCC.Text <> '' then Addressees := Addressees + ', '+ eCC.Text;
    SMTP1.Verify(Addressees);

    {wait for completion of Verify-Text message send}
    while SMTP1.Busy do Application.ProcessMessages;

{Check global flag indicating addresses are in the correct format - if
true, the text part of the message has been sent}
    if not RecvVerified then
    begin
      MessageDlg('Incorrect address format', mtError, [mbOK], 0);
      Exit;
```

```
    end
    else
      if cbSendFile.Checked then SendFile(OpenDialog.Filename);
  end
  else
    MessageDlg('Not connected to SMTP server', mtError, [mbOK], 0);
end;

{SMTP component will call this event every time its connection state
changes}
procedure TMainForm.SMTP1StateChanged(Sender: TObject; State: Smallint);
begin
  case State of
    prcConnecting:
      ConnectStatus.SimpleText := 'Connecting to SMTP server:
        '+SMTP1.RemoteHost+'...';
    prcResolvingHost:
      ConnectStatus.SimpleText := 'Resolving Host';
    prcHostResolved:
      ConnectStatus.SimpleText := 'Host Resolved';
    prcConnected:
      begin
        ConnectStatus.SimpleText := 'Connected to SMTP server:
                '+SMTP1.RemoteHost;
        SMTPConnectBtn.Caption := 'Disconnect';
      end;
    prcDisconnecting:
      ConnectStatus.SimpleText := 'Disconnecting from SMTP server:
        '+SMTP1.RemoteHost+'...';
    prcDisconnected:
      begin
        ConnectStatus.SimpleText := 'Disconnected from SMTP server:
                '+SMTP1.RemoteHost;
        SMTPConnectBtn.Caption := 'Connect';
      end;
   end;
   eSMTPServer.Enabled := not (State = prcConnected);
   eHomeAddr.Enabled := not (State = prcConnected);
end;

{The DocInput event is called each time the DocInput state changes during a
mail transfer. DocInput holds all the information about the current
transfer, including the headers, the number of bytes transferred, and the
message data itself. Although not shown in this example, you may call
DocInput's SetData method if DocInput.State = icDocData to encode the data
before each block is sent.}
procedure TMainForm.SMTP1DocInput(Sender: TObject;const DocInput: Variant);
begin
  case DocInput.State of
    icDocBegin:
      SMTPStatus.SimpleText := 'Initiating document transfer';
    icDocHeaders:
      SMTPStatus.SimpleText := 'Sending headers';
    icDocData:
      if DocInput.BytesTotal > 0 then
        SMTPStatus.SimpleText:=Format('Sending data:%d of %d bytes (%d%%)',
          [Trunc(DocInput.BytesTransferred), Trunc(DocInput.BytesTotal),
           Trunc(DocInput.BytesTransferred/DocInput.BytesTotal*100)])
      else
        SMTPStatus.SimpleText := 'Sending...';
    icDocEnd:
```

```
      if SMTPError then
        SMTPStatus.SimpleText := 'Transfer aborted'
      else
        SMTPStatus.SimpleText := Format('Mail sent to %s (%d bytes data)',
                [eTo.Text, Trunc(DocInput.BytesTransferred)]);
  end;
  SMTPStatus.Update;
end;

{The Error event is called whenever an error occurs in the background
processing. In addition to providing an error code and brief description,
you can also access the SMTP component's Errors property (of type icErrors,
an OLE object) to get more detailed information}
procedure TMainForm.SMTP1Error(Sender: TObject; Number: Smallint;
  var Description: string; Scode: Integer; const Source, HelpFile: string;
  HelpContext: Integer; var CancelDisplay: Wordbool);
var
  I: Integer;
  ErrorStr: string;
begin
  SMTPError := True;
  CancelDisplay := True;
  {Get extended error information}
  for I := 1 to SMTP1.Errors.Count do
    ErrorStr := Format(#13'(%s)', [SMTP1.Errors.Item(I).Description]);
  {Display error code, short and long error description}
  MessageDlg(Format('%d - %s%s', [Number, Description, Trim(ErrorStr)]),
        mtError, [mbOK], 0);
end;

{Unlike POP, SMTP does not require a user account on the host machine, so
no user authorization is necessary}
procedure TMainForm.SMTPConnectBtnClick(Sender: TObject);
begin
  if SMTP1.State = prcConnected then SMTP1.Quit
  else
  if SMTP1.State = prcDisconnected then
    begin
      SMTP1.RemoteHost := eSMTPServer.Text;
      SMTPError := False;
      SMTP1.Connect(NoParam, NoParam);
   end;
end;

{Unlike SMTP, users must be authorized on the POP server. The component
defines a special protocol state, popAuthorization, when it requests
authorization. If authorization is successful, the protocol state changes
to popTransaction and POP commands can be issued. Note that server
connection is independent of the authorization state.}
procedure TMainForm.POP1ProtocolStateChanged(Sender: TObject;
  ProtocolState: Smallint);
begin
  case ProtocolState of
    popAuthorization:
      POP1.Authenticate(POP1.UserID, POP1.Password);
    popTransaction:
      ConnectStatus.SimpleText := Format('User %s authorized on server %s',
        [eUsername.Text, ePOPServer.Text]);
  end;
end;
```

```
{This event is called every time the connection status of the POP server
changes}
procedure TMainForm.POP1StateChanged(Sender: TObject; State: Smallint);
begin
  case State of
    prcConnecting:
      ConnectStatus.SimpleText := 'Connecting to POP server:
        '+POP1.RemoteHost+'...';
    prcResolvingHost:
      ConnectStatus.SimpleText := 'Resolving Host';
    prcHostResolved:
      ConnectStatus.SimpleText := 'Host Resolved';
    prcConnected:
      begin
        ConnectStatus.SimpleText := 'Connected to POP server:
                '+POP1.RemoteHost;
        POPConnectBtn.Caption := 'Disconnect';
      end;
    prcDisconnecting:
      ConnectStatus.SimpleText := 'Disconnecting from POP server:
        '+POP1.RemoteHost+'...';
    prcDisconnected:
      begin
        ConnectStatus.SimpleText := 'Disconnected from POP server:
                '+POP1.RemoteHost;
        POPConnectBtn.Caption := 'Connect';
      end;
   end;
   ePOPServer.Enabled := not (State = prcConnected);
   eUsername.Enabled := not (State = prcConnected);
   ePassword.Enabled := not (State = prcConnected);
end;

{The Error event is called whenever an error occurs in the background
processing. In addition to providing an error code and brief description,
you can also access the POP component's Errors property (of type icErrors,
an OLE object) to get more detailed information}
procedure TMainForm.POP1Error(Sender: TObject; Number: Smallint;
  var Description: string; Scode: Integer; const Source, HelpFile: string;
  HelpContext: Integer; var CancelDisplay: Wordbool);
var
  I: Integer;
  ErrorStr: string;
begin
  POPError := True;
  CancelDisplay := True;
  if POP1.ProtocolState = popAuthorization then
    ConnectStatus.SimpleText := 'Authorization error';
  {Get extended error information}
  for I := 1 to POP1.Errors.Count do
    ErrorStr := Format(#13'(%s)', [POP1.Errors.Item(I).Description]);

  {Display error code, short and long error description}
  MessageDlg(Format('%d - %s%s', [Number, Description, Trim(ErrorStr)]),
        mtError, [mbOK], 0);
end;
```

```
{POP requires a valid user account on the host machine}
procedure TMainForm.POPConnectBtnClick(Sender: TObject);
begin
  if (POP1.State = prcConnected) and (POP1.ProtocolState = popTransaction)
  and not POP1.Busy then
  begin
    mReadMessage.Lines.Clear;
    POP1.Quit;
  end
  else
    if POP1.State = prcDisconnected then
    begin
      POP1.RemoteHost := ePOPServer.Text;
      POP1.UserID := eUserName.Text;
      POP1.Password := ePassword.Text;
      POP1.Connect(NoParam, NoParam);
    end;
end;

{The DocOutput event is the just like the DocInput event in 'reverse'. It
is called each time the component's DocOutput state changes during
retrieval of mail from the server. When the state = icDocData, you can call
DocOutput.GetData to decode each data block based on the MIME content type
specified in the headers.}
procedure TMainForm.POP1DocOutput(Sender: TObject;
  const DocOutput: Variant);
var
  Buffer: Variant;
  I: Integer;
begin
  case DocOutput.State of
    icDocBegin:
      POPStatus.SimpleText := 'Initiating document transfer';
    icDocHeaders:
      begin
        POPStatus.SimpleText := 'Retrieving headers';
        for I := 1 to DocOutput.Headers.Count do
          mReadMessage.Lines.Add(DocOutput.Headers.Item(I).Name+': '+
            DocOutput.Headers.Item(I).Value);
      end;
    icDocData:
      begin
        POPStatus.SimpleText := Format('Retrieving data - %d bytes',
            [Trunc(DocOutput.BytesTransferred)]);
        DocOutput.GetData(Buffer);
        mReadMessage.Text := mReadMessage.Text + Buffer;
      end;
    icDocEnd:
      if POPError then
        POPStatus.SimpleText := 'Transfer aborted'
      else
        POPStatus.SimpleText := Format('Retrieval complete (%d bytes
                data)', [Trunc(DocOutput.BytesTransferred)]);
  end;
  POPStatus.Update;
end;
```

```
{Retrieve message from the server}
procedure TMainForm.udCurMessageClick(Sender: TObject; Button: TUDBtnType);
begin
  if (POP1.State = prcConnected) and (POP1.ProtocolState = popTransaction)
then
  begin
    POPError := False;
    mReadMessage.Lines.Clear;
    POP1.RetrieveMessage(udCurMessage.Position);
  end;
end;

{The RefreshMessageCount event is called whenever the RefreshMessageCount
method is called, and also when a connection to the POP server is first
made}
procedure TMainForm.POP1RefreshMessageCount(Sender: TObject;
  Number: Integer);
begin
  FMessageCount := Number;
  udCurMessage.Max := Number;
  udCurMessage.Enabled := Number <> 0;
  lMessageCount.Caption := IntToStr(Number);
  if Number > 0 then
  begin
    udCurMessage.Min := 1;
    udCurMessage.Position := 1;
    POP1.RetrieveMessage(udCurMessage.Position);
  end;
end;

end.
```

SMTP - Simple Mail Transfer Protocol

The SMTP Client Control implements the basic client SMTP Protocol as specified by RFC821, Simple Mail Transfer Protocol. It is used to send Internet mail messages to SMTP servers.

The SMTP Control can be used for developing applications that communicate with SMTP servers to send mail messages. It provides a reusable component that gives applications access to SMTP mail servers and mail posting capabilities.

The SMTP Client Control supports a high level interface, that incorporates all SMTP commands used in sending out a mail message. Using this interface, a mail message can be sent with a single call.

Properties:
Busy Indicates a command is in progress.
DocInput Object describing input information for the document being transferred.
EnableTimer Boolean property to enable timer for the specified event.
Value is specified in the TimeOut property.
Errors A collection of errors that can be accessed for details about the last error that occurred. This collection should be used within an Error event if information passed through the Error event is not sufficient.
NotificationMode Determines when notification is issued for incoming data. Notification can also be suspended.

0 = COMPLETE: notification is provided when there is a complete response.
1 = CONTINUOUS: an event is repeatedly activated when new data arrives from the connection.

ProtocolState This property specifies the current state of the protocol. prcBase = 0 Base state before connection to server is established. prcTransaction = 1 Connection to server is established. This is the valid state for calling methods on the control.
ProtocolStateString String representation of ProtocolState.
RemoteHost The remote machine to connect to if the remoteHost parameter in the Connect method is missing. You can either provide a host name or an IP address string in dotted format. For example, 127.0.0.1.
RemotePort The remote port number to which to connect.
ReplyCode The value of the reply code is a protocol specific number that determines the result of the last request, as returned in the ReplyString property. See RFC821 for a list of valid reply codes.
ReplyString Lists the last reply string sent by the SMTP Server to the client as a result of a request.
State This property specifies the connection state of the control.

prcConnecting = 1 - Connecting. Connect has been requested, waiting for connect acknowledge.
prcResolvingHost = 2 - Resolving Host. Occurs when RemoteHost is in name format rather than dot-delimited IP format.
prcHostResolved = 3 - Resolved the host. Occurs only if ResolvingHost state has been entered previously.
prcConnected = 4 - Connection established.

prcDisconnecting = 5 - Connection closed. Disconnect has been initiated.
prcDisconnected = 6 - Initial state when protocol object is instantiated, before Connect has been initiated, after a Connect attempt failed or after Disconnect performed.

StateString A string representation of State.
TimeOut Timeout value for the specified event.
URL URL string identifying the current document being transferred. The URL format for this control is: SMTP://host:port/

Methods:

Cancel	Initiates a Cancel request to cancel a pending request. If successful, the Cancel event is called. In case of an error, the Error event is called.
Connect	Initiates a connect request. If successful, the StateChanged is called if connection is established. In case of an error, the Error event is called.
Expand	Initiates a EXPN request. If successful, Verify event will be called when the request completes. ReplyString will contain the reply from the server. In case of an error, the Error event is called.
Help	Initiates a HELP request. If successful, the Help event will be called when the request completes. ReplyString will contain the reply from the server. In case of an error, the Error event is called.
Noop	Initiates a NOOP request. Noop event will be called. In case of error, the Error event will be called. Noop verify that the connection is alive.
Quit	Initiates a Quit request to Quit the session and disconnect. In case of an error, the Error event is called.
Reset	Initiates a RSET request. If successful, the Reset event will be called. In case of an error, the Error event is called.
SendDoc	Initiates a Request to send a document identified by a URL. In case of an error, the Error event is called. The SendDoc method makes it possible to send a document. For the SMTP Control this means sending a mail message to the server.
Verify	Initiates a VRFY request. If successful, the Verify event will be called when the request completes. ReplyString will contain the reply from the server. In case of an error, the Error event is called.

Events:

OnBusy	This event is activated when a command is in progress or when a command has completed. Indicates whether or not a command is in progress.
OnCancel	This event is activated after a cancellation request has been completed and satisfied. After this event the object's state changes to idle.
OnDocInput	A DocInput related event that indicates the input data has been transferred. The DocInput event can be used in its basic form for notification during transfer.
OnError	This event is activated when an error occurs in background processing (for example, failed to connect or failed to send or receive in the background). SMTP Error Codes : The following error codes apply only to the SMTP ActiveX Control. Error Number — Error Message 2302 — Can't create temporary mail file. 2303 — Unable to send mail.
OnExpand	This event is activated after the successful completion of a Expand request.
OnHelp	This event is activated after the successful completion of a Help request.
OnNoop	This event is activated after the successful completion of a Noop request.
OnProtocolStateChanged	This event is activated whenever the protocol state changes.
OnReset	This event is activated after the successful completion of a Reset request.
OnStateChanged	This event is activated whenever the state of the transport state changes.
OnTimeOut	This event is activated when the timer for the specified event expires.

As Declared in ISP.PAS:

```
{ TSMTP }

TSMTPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
TSMTPTimeout = procedure(Sender: TObject; event: Smallint; var Continue:
TOleBool) of object;
TSMTPStateChanged = procedure(Sender: TObject; State: Smallint) of object;
TSMTPProtocolStateChanged = procedure(Sender: TObject; ProtocolState:
Smallint) of object;
TSMTPBusy = procedure(Sender: TObject; isBusy: TOleBool) of object;
TSMTPDocInput = procedure(Sender: TObject; const DocInput: Variant) of
object;
```

```
TSMTP = class(TOleControl)
  private
    FOnError: TSMTPError;
    FOnTimeout: TSMTPTimeout;
    FOnCancel: TNotifyEvent;
    FOnStateChanged: TSMTPStateChanged;
    FOnProtocolStateChanged: TSMTPProtocolStateChanged;
    FOnBusy: TSMTPBusy;
    FOnLog: TNotifyEvent;
    FOnDocInput: TSMTPDocInput;
    FOnReset: TNotifyEvent;
    FOnVerify: TNotifyEvent;
    FOnExpand: TNotifyEvent;
    FOnHelp: TNotifyEvent;
    FOnNoop: TNotifyEvent;
    function Get_Timeout(event: Smallint): Integer; stdcall;
    procedure Set_Timeout(event: Smallint; Value: Integer); stdcall;
    procedure Set_EnableTimer(event: Smallint; Value: TOleBool); stdcall;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure Cancel; stdcall;
    procedure Connect(const RemoteHost, RemotePort: Variant); stdcall;
    procedure SendDoc(const URL, Headers, InputData, InputFile, OutputFile:
        Variant); stdcall;
    procedure Reset; stdcall;
    procedure Verify(const name: string); stdcall;
    procedure Expand(const name: string); stdcall;
    procedure Help(const helpTopic: Variant); stdcall;
    procedure Noop; stdcall;
    procedure Quit; stdcall;
    property State: Smallint index 503 read GetSmallintProp;
    property ProtocolState: Smallint index 504 read GetSmallintProp;
    property ReplyString: string index 505 read GetStringProp;
    property ReplyCode: Integer index 506 read GetIntegerProp;
    property Errors: Variant index 508 read GetVariantProp;
    property Busy: TOleBool index 509 read GetOleBoolProp;
    property StateString: string index 511 read GetStringProp;
    property ProtocolStateString: string index 512 read GetStringProp;
    property DocInput: Variant index 1002 read GetVariantProp;
    property Timeout[event: Smallint]: Integer read Get_Timeout write
        Set_Timeout;
    property EnableTimer[event: Smallint]: TOleBool write Set_EnableTimer;
  published
    property RemoteHost: string index 0 read GetStringProp write
        SetStringProp stored False;
    property RemotePort: Integer index 502 read GetIntegerProp write
        SetIntegerProp stored False;
    property NotificationMode: Smallint index 510 read GetSmallintProp
        write SetSmallintProp stored False;
    property Logging: TOleBool index 514 read GetOleBoolProp write
        SetOleBoolProp stored False;
    property URL: string index 1001 read GetStringProp write SetStringProp
        stored False;
    property OnError: TSMTPError read FOnError write FOnError;
    property OnTimeout: TSMTPTimeout read FOnTimeout write FOnTimeout;
    property OnCancel: TNotifyEvent read FOnCancel write FOnCancel;
    property OnStateChanged: TSMTPStateChanged read FOnStateChanged write
        FOnStateChanged;
```

```
    property OnProtocolStateChanged: TSMTPProtocolStateChanged read
        FOnProtocolStateChanged write FOnProtocolStateChanged;
    property OnBusy: TSMTPBusy read FOnBusy write FOnBusy;
    property OnLog: TNotifyEvent read FOnLog write FOnLog;
    property OnDocInput: TSMTPDocInput read FOnDocInput write FOnDocInput;
    property OnReset: TNotifyEvent read FOnReset write FOnReset;
    property OnVerify: TNotifyEvent read FOnVerify write FOnVerify;
    property OnExpand: TNotifyEvent read FOnExpand write FOnExpand;
    property OnHelp: TNotifyEvent read FOnHelp write FOnHelp;
    property OnNoop: TNotifyEvent read FOnNoop write FOnNoop;
  end;
```

Example:

```
unit Main;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls, OleCtrls,
  ISP;

type

  TMainForm = class(TForm)
    OpenDialog: TOpenDialog;
    SMTP1: TSMTP;
    POP1: TPOP;
    PageControl1: TPageControl;
    SendPage: TTabSheet;
    RecvPage: TTabSheet;
    ConPage: TTabSheet;
    Panel1: TPanel;
    Label1: TLabel;
    Label3: TLabel;
    Label2: TLabel;
    eTo: TEdit;
    eCC: TEdit;
    eSubject: TEdit;
    SendBtn: TButton;
    ClearBtn: TButton;
    reMessageText: TRichEdit;
    SMTPStatus: TStatusBar;
    Panel3: TPanel;
    mReadMessage: TMemo;
    POPStatus: TStatusBar;
    cbSendFile: TCheckBox;
    GroupBox1: TGroupBox;
    ePOPServer: TEdit;
    Label6: TLabel;
    Label5: TLabel;
    eUserName: TEdit;
    ePassword: TEdit;
    Label4: TLabel;
    GroupBox2: TGroupBox;
    Label7: TLabel;
    eSMTPServer: TEdit;
    SMTPConnectBtn: TButton;
    POPConnectBtn: TButton;
    eHomeAddr: TEdit;
    Label8: TLabel;
    Panel2: TPanel;
```

```
    Label9: TLabel;
    lMessageCount: TLabel;
    Label10: TLabel;
    eCurMessage: TEdit;
    udCurMessage: TUpDown;
    ConnectStatus: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure POP1StateChanged(Sender: TObject; State: Smallint);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure SMTP1StateChanged(Sender: TObject; State: Smallint);
    procedure SMTP1DocInput(Sender: TObject; const DocInput: Variant);
    procedure SMTP1Error(Sender: TObject; Number: Smallint;
      var Description: string; Scode: Integer; const Source,
      HelpFile: string; HelpContext: Integer; var CancelDisplay: Wordbool);
    procedure FormResize(Sender: TObject);
    procedure ClearBtnClick(Sender: TObject);
    procedure SMTP1Verify(Sender: TObject);
    procedure SendBtnClick(Sender: TObject);
    procedure POP1ProtocolStateChanged(Sender: TObject;
      ProtocolState: Smallint);
    procedure POP1Error(Sender: TObject; Number: Smallint;
      var Description: string; Scode: Integer; const Source,
      HelpFile: string; HelpContext: Integer; var CancelDisplay: Wordbool);
    procedure SMTPConnectBtnClick(Sender: TObject);
    procedure POPConnectBtnClick(Sender: TObject);
    procedure eSMTPServerChange(Sender: TObject);
    procedure ePOPServerChange(Sender: TObject);
    procedure cbSendFileClick(Sender: TObject);
    procedure POP1DocOutput(Sender: TObject; const DocOutput: Variant);
    procedure udCurMessageClick(Sender: TObject; Button: TUDBtnType);
    procedure POP1RefreshMessageCount(Sender: TObject; Number: Integer);
  private
    RecvVerified,
    SMTPError,
    POPError: Boolean;
    FMessageCount: Integer;
    procedure SendFile(Filename: string);
    procedure SendMessage;
    procedure CreateHeaders;
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

uses OLEAuto;

const
  icDocBegin = 1;
  icDocHeaders = 2;
  icDocData = 3;
  icDocEnd = 5;

{When calling a component method which maps onto an OLE call, NoParam
substitutes for an optional parameter. As an alternative to calling the
component method, you may access the component's OLEObject directly -
i.e., Component.OLEObject.MethodName(,Foo,,Bar)}
function NoParam: Variant;
begin
```

```
  TVarData(Result).VType := varError;
  TVarData(Result).VError := DISP_E_PARAMNOTFOUND;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  SMTPError := False;
  POPError := False;
  FMessageCount := 0;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if POP1.State = prcConnected then POP1.Quit;
  if SMTP1.State = prcConnected then SMTP1.Quit;
end;

procedure TMainForm.FormResize(Sender: TObject);
begin
  SendBtn.Left := ClientWidth - SendBtn.Width - 10;
  ClearBtn.Left := ClientWidth - ClearBtn.Width - 10;
  cbSendFile.Left := ClientWidth - cbSendFile.Width - 10;
  eTo.Width := SendBtn.Left - eTo.Left - 10;
  eCC.Width := SendBtn.Left - eCC.Left - 10;
  eSubject.Width := SendBtn.Left - eSubject.Left - 10;
end;

procedure TMainForm.ClearBtnClick(Sender: TObject);
begin
  eTo.Text := '';
  eCC.Text := '';
  eSubject.Text := '';
  OpenDialog.Filename := '';
  reMessageText.Lines.Clear;
end;

procedure TMainForm.eSMTPServerChange(Sender: TObject);
begin
  SMTPConnectBtn.Enabled:=(eSMTPServer.Text<>'') and (eHomeAddr.Text<> '');
end;

procedure TMainForm.ePOPServerChange(Sender: TObject);
begin
  POPConnectBtn.Enabled := (ePOPServer.Text <> '') and (eUsername.Text<>'')
    and (ePassword.Text <> '');
end;

procedure TMainForm.cbSendFileClick(Sender: TObject);
begin
  if cbSendFile.Checked then
  begin
    if OpenDialog.Execute then
      cbSendFile.Caption := cbSendFile.Caption + ': '+OpenDialog.Filename
    else
      cbSendFile.Checked := False;
  end else
    cbSendFile.Caption := '&Attach Text File';
end;
```

```
{Clear and repopulate MIME headers, using the component's DocInput
property. A separate DocInput OLE object could also be used. See RFC 1521 /
RFC 1522 for complete information on MIME types.}
procedure TMainForm.CreateHeaders;
begin
  with SMTP1 do
  begin
    DocInput.Headers.Clear;
    DocInput.Headers.Add('To', eTo.Text);
    DocInput.Headers.Add('From', eHomeAddr.Text);
    DocInput.Headers.Add('CC', eCC.Text);
    DocInput.Headers.Add('Subject', eSubject.Text);
    DocInput.Headers.Add('Message-Id', Format('%s_%s_%s',
        [Application.Title, DateTimeToStr(Now), eHomeAddr.Text]));
    DocInput.Headers.Add('Content-Type', 'TEXT/PLAIN charset=US-ASCII');
  end;
end;

{Send a simple mail message}
procedure TMainForm.SendMessage;
begin
  CreateHeaders;
  with SMTP1 do
    SendDoc(NoParam, DocInput.Headers, reMessageText.Text, '', '');
end;

{Send a disk file. Leave SendDoc's InputData parameter blank and specify a
filename for InputFile to send the contents of a disk file. You can use the
DocInput event and GetData methods to do custom encoding (Base64, UUEncode,
etc.) }
procedure TMainForm.SendFile(Filename: string);
begin
  CreateHeaders;
  with SMTP1 do
  begin
    DocInput.Filename := FileName;
    SendDoc(NoParam, DocInput.Headers, NoParam, DocInput.FileName, '');
  end;
end;

{Set global flag indicating recipients are addressable (this only ensures
that the address is in the correct format, not that it exists and is
deliverable), then send the text part of the message}
procedure TMainForm.SMTP1Verify(Sender: TObject);
begin
  SendMessage;
  RecvVerified := True;
end;

{Verify addressees, send text message in the Verify event, and if an
attachment is specified, send it}
procedure TMainForm.SendBtnClick(Sender: TObject);
var
  Addressees: string;
begin
  if SMTP1.State = prcConnected then
  begin
    RecvVerified := False;
    SMTPError := False;
    Addressees := eTo.Text;
```

```
    if eCC.Text <> '' then Addressees := Addressees + ', '+ eCC.Text;
    SMTP1.Verify(Addressees);

    {wait for completion of Verify-Text message send}
    while SMTP1.Busy do Application.ProcessMessages;

{Check global flag indicating addresses are in the correct format - if
true, the text part of the message has been sent}
    if not RecvVerified then
    begin
      MessageDlg('Incorrect address format', mtError, [mbOK], 0);
      Exit;
    end
    else
      if cbSendFile.Checked then SendFile(OpenDialog.Filename);
  end
  else
    MessageDlg('Not connected to SMTP server', mtError, [mbOK], 0);
end;

{SMTP component will call this event every time its connection state
changes}
procedure TMainForm.SMTP1StateChanged(Sender: TObject; State: Smallint);
begin
  case State of
    prcConnecting:
      ConnectStatus.SimpleText := 'Connecting to SMTP server:
        '+SMTP1.RemoteHost+'...';
    prcResolvingHost:
      ConnectStatus.SimpleText := 'Resolving Host';
    prcHostResolved:
      ConnectStatus.SimpleText := 'Host Resolved';
    prcConnected:
      begin
        ConnectStatus.SimpleText := 'Connected to SMTP server:
                '+SMTP1.RemoteHost;
        SMTPConnectBtn.Caption := 'Disconnect';
      end;
    prcDisconnecting:
      ConnectStatus.SimpleText := 'Disconnecting from SMTP server:
        '+SMTP1.RemoteHost+'...';
    prcDisconnected:
      begin
        ConnectStatus.SimpleText := 'Disconnected from SMTP server:
                '+SMTP1.RemoteHost;
        SMTPConnectBtn.Caption := 'Connect';
      end;
   end;
   eSMTPServer.Enabled := not (State = prcConnected);
   eHomeAddr.Enabled := not (State = prcConnected);
end;

{The DocInput event is called each time the DocInput state changes during a
mail transfer. DocInput holds all the information about the current
transfer, including the headers, the number of bytes transferred, and the
message data itself. Although not shown in this example, you may call
DocInput's SetData method if DocInput.State = icDocData to encode the data
before each block is sent.}
procedure TMainForm.SMTP1DocInput(Sender: TObject;
  const DocInput: Variant);
begin
```

```
  case DocInput.State of
    icDocBegin:
      SMTPStatus.SimpleText := 'Initiating document transfer';
    icDocHeaders:
      SMTPStatus.SimpleText := 'Sending headers';
    icDocData:
      if DocInput.BytesTotal > 0 then
        SMTPStatus.SimpleText := Format('Sending data: %d of %d bytes
            (%d%%)', [Trunc(DocInput.BytesTransferred),
            Trunc(DocInput.BytesTotal),
            Trunc(DocInput.BytesTransferred/DocInput.BytesTotal*100)])
      else
        SMTPStatus.SimpleText := 'Sending...';
    icDocEnd:
      if SMTPError then
        SMTPStatus.SimpleText := 'Transfer aborted'
      else
        SMTPStatus.SimpleText := Format('Mail sent to %s (%d bytes data)',
                [eTo.Text, Trunc(DocInput.BytesTransferred)]);
  end;
  SMTPStatus.Update;
end;

{The Error event is called whenever an error occurs in the background
processing. In addition to providing an error code and brief description,
you can also access the SMTP component's Errors property (of type icErrors,
an OLE object) to get more detailed information}
procedure TMainForm.SMTP1Error(Sender: TObject; Number: Smallint;
  var Description: string; Scode: Integer; const Source, HelpFile: string;
  HelpContext: Integer; var CancelDisplay: Wordbool);
var
  I: Integer;
  ErrorStr: string;
begin
  SMTPError := True;
  CancelDisplay := True;
  {Get extended error information}
  for I := 1 to SMTP1.Errors.Count do
    ErrorStr := Format(#13'(%s)', [SMTP1.Errors.Item(I).Description]);
  {Display error code, short and long error description}
  MessageDlg(Format('%d - %s%s', [Number, Description, Trim(ErrorStr)]),
        mtError, [mbOK], 0);
end;

{Unlike POP, SMTP does not require a user account on the host machine, so
no user authorization is necessary}
procedure TMainForm.SMTPConnectBtnClick(Sender: TObject);
begin
  if SMTP1.State = prcConnected then SMTP1.Quit
  else
  if SMTP1.State = prcDisconnected then
    begin
      SMTP1.RemoteHost := eSMTPServer.Text;
      SMTPError := False;
      SMTP1.Connect(NoParam, NoParam);
   end;
end;
```

```
{Unlike SMTP, users must be authorized on the POP server. The component
defines a special protocol state, popAuthorization, when it requests
authorization. If authorization is successful, the protocol state changes
to popTransaction and POP commands can be issued. Note that server
connection is independent of the authorization state.}
procedure TMainForm.POP1ProtocolStateChanged(Sender: TObject;
  ProtocolState: Smallint);
begin
  case ProtocolState of
    popAuthorization:
      POP1.Authenticate(POP1.UserID, POP1.Password);
    popTransaction:
      ConnectStatus.SimpleText := Format('User %s authorized on server %s',
        [eUsername.Text, ePOPServer.Text]);
  end;
end;

{This event is called every time the connection status of the POP server
changes}
procedure TMainForm.POP1StateChanged(Sender: TObject; State: Smallint);
begin
  case State of
    prcConnecting:
      ConnectStatus.SimpleText := 'Connecting to POP server:
        '+POP1.RemoteHost+'...';
    prcResolvingHost:
      ConnectStatus.SimpleText := 'Resolving Host';
    prcHostResolved:
      ConnectStatus.SimpleText := 'Host Resolved';
    prcConnected:
      begin
        ConnectStatus.SimpleText := 'Connected to POP server:
                '+POP1.RemoteHost;
        POPConnectBtn.Caption := 'Disconnect';
      end;
    prcDisconnecting:
      ConnectStatus.SimpleText := 'Disconnecting from POP server:
        '+POP1.RemoteHost+'...';
    prcDisconnected:
      begin
        ConnectStatus.SimpleText := 'Disconnected from POP server:
                '+POP1.RemoteHost;
        POPConnectBtn.Caption := 'Connect';
      end;
   end;
   ePOPServer.Enabled := not (State = prcConnected);
   eUsername.Enabled := not (State = prcConnected);
   ePassword.Enabled := not (State = prcConnected);
end;

{The Error event is called whenever an error occurs in the background
processing. In addition to providing an error code and brief description,
you can also access the POP component's Errors property (of type icErrors,
an OLE object) to get more detailed information}
procedure TMainForm.POP1Error(Sender: TObject; Number: Smallint;
  var Description: string; Scode: Integer; const Source, HelpFile: string;
  HelpContext: Integer; var CancelDisplay: Wordbool);
  var
  I: Integer;
  ErrorStr: string;
begin
```

```
  POPError := True;
  CancelDisplay := True;
  if POP1.ProtocolState = popAuthorization then
    ConnectStatus.SimpleText := 'Authorization error';

  {Get extended error information}
  for I := 1 to POP1.Errors.Count do
    ErrorStr := Format(#13'(%s)', [POP1.Errors.Item(I).Description]);

  {Display error code, short and long error description}
  MessageDlg(Format('%d - %s%s', [Number, Description, Trim(ErrorStr)]),
        mtError, [mbOK], 0);
end;

{POP requires a valid user account on the host machine}
procedure TMainForm.POPConnectBtnClick(Sender: TObject);
begin
  if (POP1.State = prcConnected) and (POP1.ProtocolState = popTransaction)
  and not POP1.Busy then
  begin
    mReadMessage.Lines.Clear;
    POP1.Quit;
  end
  else
    if POP1.State = prcDisconnected then
    begin
      POP1.RemoteHost := ePOPServer.Text;
      POP1.UserID := eUserName.Text;
      POP1.Password := ePassword.Text;
      POP1.Connect(NoParam, NoParam);
    end;
end;

{The DocOutput event is the just like the DocInput event in 'reverse'. It
is called each time the component's DocOutput state changes during
retrieval of mail from the server. When the state = icDocData, you can call
DocOutput.GetData to decode each data block based on the MIME content type
specified in the headers.}
procedure TMainForm.POP1DocOutput(Sender: TObject;
  const DocOutput: Variant);
var
  Buffer: Variant;
  I: Integer;
begin
  case DocOutput.State of
    icDocBegin:
      POPStatus.SimpleText := 'Initiating document transfer';
    icDocHeaders:
      begin
        POPStatus.SimpleText := 'Retrieving headers';
        for I := 1 to DocOutput.Headers.Count do
          mReadMessage.Lines.Add(DocOutput.Headers.Item(I).Name+': '+
            DocOutput.Headers.Item(I).Value);
      end;
    icDocData:
      begin
        POPStatus.SimpleText := Format('Retrieving data - %d bytes',
            [Trunc(DocOutput.BytesTransferred)]);
        DocOutput.GetData(Buffer);
        mReadMessage.Text := mReadMessage.Text + Buffer;
      end;
```

```
    icDocEnd:
      if POPError then
        POPStatus.SimpleText := 'Transfer aborted'
      else
        POPStatus.SimpleText:=Format('Retrieval complete (%d bytes data)',
          [Trunc(DocOutput.BytesTransferred)]);
  end;
  POPStatus.Update;
end;

{Retrieve message from the server}
procedure TMainForm.udCurMessageClick(Sender: TObject; Button: TUDBtnType);
begin
  if (POP1.State = prcConnected) and (POP1.ProtocolState = popTransaction)
then
  begin
    POPError := False;
    mReadMessage.Lines.Clear;
    POP1.RetrieveMessage(udCurMessage.Position);
  end;
end;

{The RefreshMessageCount event is called whenever the RefreshMessageCount
method is called, and also when a connection to the POP server is first
made}
procedure TMainForm.POP1RefreshMessageCount(Sender: TObject;
  Number: Integer);
begin
  FMessageCount := Number;
  udCurMessage.Max := Number;
  udCurMessage.Enabled := Number <> 0;
  lMessageCount.Caption := IntToStr(Number);
  if Number > 0 then
  begin
    udCurMessage.Min := 1;
    udCurMessage.Position := 1;
    POP1.RetrieveMessage(udCurMessage.Position);
  end;
end;

end.
```

TCP - Transmission Control Protocol

The WinSock TCP ActiveX Control implements the WinSock Transmission Control Protocol (TCP) for both client and server applications.

Invisible to the user, the TCP Control provides easy access to TCP network services. By setting properties and calling methods on the control, you can easily connect to a remote machine and exchange data in both directions. Events are used to notify you of network activities.

For further reference material on TCP, see RFC 1001 / RFC 1002

Properties:

BytesReceived	Advanced property. It shows the amount of data received currently in the receive buffer). The GetData method should be used to retrieve data.
LocalHostName	Local machine name.
LocalIP	The IP address of the local machine. It has the format: number.number.number.number
LocalPort	For the client, this designates the local port to use. Specify port 0 if the application does not need a specific port. In this case, the control will select a random port. After a connection is established, this is the local port used for the TCP connection. For the server, this is the local port to listen on. If port 0 is specified, a random port is used. After calling the Listen method, the property contains the actual port that has been selected.
RemoteHost	The remote machine to connect to if the RemoteHost parameter of the Connect method is not specified. You can either provide a host name or an IP address string in dotted format.
RemoteHostIP	For the client, after a connection has been established (i.e., after the Connect event has been activated), this property contains the IP string of the remote machine in dotted format. For server, after an incoming connection request (ConnectionRequest event), this property contains the IP string (in dotted format) of the remote machine initiating the connection.
RemotePort	For the client, this is the remote port number to which to connect if the RemotePort parameter of the Connect method is not specified.

	For the server, after an incoming connection request event, ConnectionRequest has been activated) this property contains the port that the remote machine uses to connect to this server.
SocketHandle	This is the socket handle the control uses to communicate with the WinSock layer.
State	The state of the control, expressed as an enum type.

sckClosed	= 0	Closed
sckOpen	= 1	Open
sckListening	= 2	Listening
sckConnectionPending	= 3	Connection pending
sckResolvingHost	= 4	Resolving host
sckHostResolved	= 5	Host resolved
sckConnecting	= 6	Connecting
sckConnected	= 7	Connected
sckClosing	= 8	Peer is closing the connection
sckError	= 9	Error

Methods:

Accept	This method is used to accept an incoming connection when handling a ConnectionRequest event. Accept should be used on a new control instance (other than the one that is in the listening state.)
Close	Closes a TCP connection or a listening socket for both client and server.
Connect	Initiates connection to remote machine.
GetData	Retrieves data.
Listen	It includes creating a socket and putting the socket in the listening mode. When there is an incoming connection, the ConnectionRequest event is activated. When handling ConnectionRequest, the application should use the Accept method (on a new control instance) to accept the connection.
PeekData	Similar to GetData except PeekData does not remove data from input queue.
SendData	Sends data to peer.

Events:

OnClose	The event is activated when the peer closes the connection. Applications should use the Close method to correctly close the connection.

OnConnect	The event is activated when a connection has been successfully established. After this event is activated, you can send or receive data on the control.
OnConnectionRequest	The event is activated when there is an incoming connection request. RemoteHostIP and RemotePort properties store the information about the client after the event is activated. The server can decide whether or not to accept the connection. If the incoming connection is not accepted, the peer (client) will get the Close event. Use the Accept method (on a new control instance) to accept an incoming connection.
OnDataArrival	The event is activated when new data arrives.
OnError	This standard error event is activated whenever an error occurs in background processing (for example, failed to connect, or failed to send or receive in the background). WinSock Error Codes : The following error codes apply to the WinSock ActiveX Controls

Error Number	Error Message
10004	The operation is canceled
10013	The requested address is a broadcast address, but flag is not set
10014	Invalid argument
10022	Socket not bound, invalid address or listen is not invoked prior to accept
10024	No more file descriptors are available, accept queue is empty
10035	Socket is non-blocking and the specified operation will block
10036	A blocking Winsock operation is in progress
10037	The operation is completed. No blocking operation is in progress.
10038	The descriptor is not a socket
10039	Destination address is required
10040	The datagram is too large to fit into the buffer and is truncated
10041	The specified port is the wrong type for this socket

	10042	Option unknown, or unsupported
	10043	The specified port is not supported
	10044	Socket type not supported in this address family
	10045	Socket is not a type that supports connection oriented service
	10047	Address Family is not supported
	10048	Address in use
	10049	Address is not available from the local machine
	10050	Network subsystem failed
	10051	The network cannot be reached from this host at this time
	10052	Connection has timed out when SO_KEEPALIVE is set
	10053	Connection is aborted due to timeout or other failure
	10054	The connection is reset by remote side
	10055	No buffer space is available
	10056	Socket is already connected
	10057	Socket is not connected
	10058	Socket has been shut down
	10060	The attempt to connect timed out
	10061	Connection is forcefully rejected
	10201	Socket already created for this object
	10202	Socket has not been created for this object
	11001	Authoritative answer: Host not found
	11002	Non-Authoritative answer: Host not found
	11003	Non-recoverable errors
	11004	Valid name, no data record of requested type
OnSendComplete	The event is activated when the send buffer is empty.	
OnSendProgress	This event notifies the user of sending progress. It is activated when more data has been accepted by the stack.	

As Declared in ISP.PAS:

```
{ TTCP }

TTCPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
TTCPDataArrival = procedure(Sender: TObject; bytesTotal: Integer) of
object;
```

```
TTCPConnectionRequest = procedure(Sender: TObject; requestID: Integer) of
object;
TTCPSendProgress = procedure(Sender: TObject; bytesSent, bytesRemaining:
Integer) of object;

TTCP = class(TOleControl)
  private
    FOnError: TTCPError;
    FOnDataArrival: TTCPDataArrival;
    FOnConnect: TNotifyEvent;
    FOnConnectionRequest: TTCPConnectionRequest;
    FOnClose: TNotifyEvent;
    FOnSendProgress: TTCPSendProgress;
    FOnSendComplete: TNotifyEvent;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure Connect(const RemoteHost, RemotePort: Variant); stdcall;
    procedure Listen; stdcall;
    procedure Accept(requestID: Integer); stdcall;
    procedure SendData(const data: Variant); stdcall;
    procedure GetData(var data: Variant; const type_, maxLen: Variant);
        stdcall;
    procedure PeekData(var data: Variant; const type_, maxLen: Variant);
        stdcall;
    procedure Close; stdcall;
    property RemoteHostIP: string index 1001 read GetStringProp;
    property LocalHostName: string index 1002 read GetStringProp;
    property LocalIP: string index 1003 read GetStringProp;
    property SocketHandle: Integer index 1004 read GetIntegerProp;
    property State: Smallint index 503 read GetSmallintProp;
    property BytesReceived: Integer index 1101 read GetIntegerProp;
  published
    property RemoteHost: string index 0 read GetStringProp write
        SetStringProp stored False;
    property RemotePort: Integer index 502 read GetIntegerProp write
        SetIntegerProp stored False;
    property LocalPort: Integer index 1010 read GetIntegerProp write
        SetIntegerProp stored False;
    property OnError: TTCPError read FOnError write FOnError;
    property OnDataArrival: TTCPDataArrival read FOnDataArrival write
        FOnDataArrival;
    property OnConnect: TNotifyEvent read FOnConnect write FOnConnect;
    property OnConnectionRequest: TTCPConnectionRequest read
        FOnConnectionRequest write FOnConnectionRequest;
    property OnClose: TNotifyEvent read FOnClose write FOnClose;
    property OnSendProgress: TTCPSendProgress read FOnSendProgress write
        FOnSendProgress;
    property OnSendComplete: TNotifyEvent read FOnSendComplete write
        FOnSendComplete;
  end;

{ TUDP }

TUDPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
TUDPDataArrival = procedure(Sender: TObject; bytesTotal: Integer) of
object;
```

```
TUDP = class(TOleControl)
  private
    FOnError: TUDPError;
    FOnDataArrival: TUDPDataArrival;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure SendData(const data: Variant); stdcall;
    procedure GetData(var data: Variant; const type_: Variant); stdcall;
    property RemoteHostIP: string index 1001 read GetStringProp;
    property LocalHostName: string index 1002 read GetStringProp;
    property LocalIP: string index 1003 read GetStringProp;
    property SocketHandle: Integer index 1004 read GetIntegerProp;
  published
    property RemoteHost: string index 0 read GetStringProp write
        SetStringProp stored False;
    property RemotePort: Integer index 502 read GetIntegerProp write
        SetIntegerProp stored False;
    property LocalPort: Integer index 1010 read GetIntegerProp write
        SetIntegerProp stored False;
    property OnError: TUDPError read FOnError write FOnError;
    property OnDataArrival: TUDPDataArrival read FOnDataArrival write
        FOnDataArrival;
  end;
```

Example:

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  OleCtrls, ISP, ExtCtrls, ComCtrls, Menus, OLE2, StdCtrls, Buttons;

type
  TChatForm = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    FileConnectItem: TMenuItem;
    FileListenItem: TMenuItem;
    StatusBar1: TStatusBar;
    Bevel1: TBevel;
    Panel1: TPanel;
    TCP1: TTCP;
    Memo1: TMemo;
    TCP2: TTCP;
    Memo2: TMemo;
    N1: TMenuItem;
    SpeedButton1: TSpeedButton;
    Disconnect1: TMenuItem;
    procedure FileListenItemClick(Sender: TObject);
    procedure TCP1ConnectionRequest(Sender: TObject; requestID: Integer);
    procedure FileConnectItemClick(Sender: TObject);
    procedure TCP1DataArrival(Sender: TObject; bytesTotal: Integer);
    procedure Exit1Click(Sender: TObject);
    procedure TCP1Close(Sender: TObject);
    procedure Memo1KeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure FormCreate(Sender: TObject);
```

```
    procedure TCP2Connect(Sender: TObject);
    procedure TCP1Error(Sender: TObject; Number: Smallint;
      var Description: string; Scode: Integer; const Source,
      HelpFile: string; HelpContext: Integer; var CancelDisplay: Wordbool);
    procedure Disconnect1Click(Sender: TObject);
    procedure TCP2Close(Sender: TObject);
  end;

var
  ChatForm: TChatForm;
  Server: String;
  N: Integer;

implementation
{$R *.DFM}

procedure TChatForm.FileListenItemClick(Sender: TObject);
begin
  FileListenItem.Checked := not FileListenItem.Checked;
  if FileListenItem.Checked then
  begin
    TCP2.Close;
    TCP1.Listen;
    Statusbar1.Panels[0].Text := 'Listening...'
  end
  else
  begin
    if TCP1.State <> sckClosed then
      TCP1.Close;
    Statusbar1.Panels[0].Text := '';
  end;
end;

procedure TChatForm.TCP1ConnectionRequest(Sender: TObject;
  requestID: Integer);
begin
  FileListenItemClick(nil);
  TCP2.Accept(requestID);
  Statusbar1.Panels[0].Text := 'Connected to: ' + TCP1.RemoteHostIP;
end;

procedure TChatForm.FileConnectItemClick(Sender: TObject);
begin
  if TCP2.State <> sckClosed then TCP2.Close;
  if InputQuery('Computer to connect to', 'Address (either IP or Name):',
        Server) then
    if Length(Server) > 0 then
      TCP2.Connect(Server, 1024);
end;

procedure TChatForm.TCP1DataArrival(Sender: TObject; bytesTotal: Integer);
var
  Data, DataType: Variant;
  Ch: Integer;
begin
  TCP2.GetData(Data, VT_BSTR, bytesTotal);
  Memo2.Lines.Add(Data);
end;
```

```
procedure TChatForm.Exit1Click(Sender: TObject);
begin
  TCP1.Close;
  TCP2.Close;
  Close;
end;

procedure TChatForm.TCP1Close(Sender: TObject);
begin
  ShowMessage('TCP1.Close');
  Statusbar1.Panels[0].Text := 'Disconnected';
end;

procedure TChatForm.Memo1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = VK_Return then
    TCP2.SendData(Memo1.Lines[Memo1.Lines.Count - 1]);
end;

procedure TChatForm.FormCreate(Sender: TObject);
begin
  FileListenItemClick(nil);
end;

procedure TChatForm.TCP2Connect(Sender: TObject);
begin
  if TCP1.State = sckListening then
    FileListenItemClick(nil);
  Statusbar1.Panels[0].Text := 'Connected to: ' + Server;
end;

procedure TChatForm.TCP1Error(Sender: TObject; Number: Smallint;
  var Description: string; Scode: Integer; const Source, HelpFile: string;
  HelpContext: Integer; var CancelDisplay: Wordbool);
begin
  ShowMessage(Description);
end;

procedure TChatForm.Disconnect1Click(Sender: TObject);
begin
  TCP2.Close;
  FileListenItemClick(nil);
end;

procedure TChatForm.TCP2Close(Sender: TObject);
begin
  TCP1.Close;
  FileListenItemClick(nil);
end;

end.
```

UDP - User Datagram Protocol

The WinSock UDP ActiveX Control implements WinSock UDP (User Datagram Protocol) for both client and server. The control represents a communication point utilizing UDP network services. It can be used to send and retrieve UDP data.

Invisible to the user, the UDP Control, provides easy access to UDP network services. It can be used by both Delphi and C++ programmers. To write UDP applications you do not need to understand the details of UDP or to call low level WinSock APIs. By setting properties and calling methods on the control, you can easily send data to a remote machine or retrieve data from the network. Events are used to notify users of network activities.

For further reference material on TCP, see RFC 1001 / RFC 1002

Properties:

LocalHostName	This property defines the local machine name.
LocalIP	The IP address of the local machine. It has the format: number.number.number.number
LocalPort	Designates the local port to use.
RemoteHost	The remote machine to which to send UDP data. You can enter either a host name or an IP address string in dotted format (for example, 156.10.5.298).
RemoteHostIP	After the GetData method, this property contains the IP address of the remote machine sending the UDP data.
RemotePort	This property specifies the remote port number on the remote machine to which UDP data is sent. After the GetData method, this property contains the remote port that is sending the UDP data.
SocketHandle	This is the socket handle the control uses to communicate with the WinSock layer. This property is for advanced programmers. You can use SocketHandle in direct WinSock API calls. However, you should be aware that if WinSock calls are used directly, certain events may not be activated appropriately.

Methods:

GetData	Retrieves data.
SendData	This method sends data to remote machine.

Events:

OnDataArrival	The event is activated when a new UDP packet arrives.

OnError

The event is activated whenever an error occurs in background processing (for example, failed to connect, or failed to send or receive in the background).

WinSock Error Codes : The following error codes apply to the WinSock ActiveX Controls

Error Number	Error Message
10004	The operation is canceled
10013	The requested address is a broadcast address, but flag is not set
10014	Invalid argument
10022	Socket not bound, invalid address or listen is not invoked prior to accept
10024	No more file descriptors are available, accept queue is empty
10035	Socket is non-blocking and the specified operation will block
10036	A blocking Winsock operation is in progress
10037	The operation is completed. No blocking operation is in progress.
10038	The descriptor is not a socket
10039	Destination address is required
10040	The datagram is too large to fit into the buffer and is truncated
10041	The specified port is the wrong type for this socket
10042	Option unknown, or unsupported
10043	The specified port is not supported
10044	Socket type not supported in this address family
10045	Socket is not a type that supports connection oriented service
10047	Address Family is not supported
10048	Address in use
10049	Address is not available from the local machine
10050	Network subsystem failed
10051	The network cannot be reached from this host at this time
10052	Connection has timed out when SO_KEEPALIVE is set
10053	Connection is aborted due to timeout or other failure

10054	The connection is reset by remote side
10055	No buffer space is available
10056	Socket is already connected
10057	Socket is not connected
10058	Socket has been shut down
10060	The attempt to connect timed out
10061	Connection is forcefully rejected
10201	Socket already created for this object
10202	Socket has not been created for this object
11001	Authoritative answer: Host not found
11002	Non-Authoritative answer: Host not found
11003	Non-recoverable errors
11004	Valid name, no data record of requested type

As Declared in ISP.PAS:

```
{ TUDP }

TUDPError = procedure(Sender: TObject; Number: Smallint; var Description:
string; Scode: Integer; const Source, HelpFile: string; HelpContext:
Integer; var CancelDisplay: TOleBool) of object;
TUDPDataArrival = procedure(Sender: TObject; bytesTotal: Integer) of
object;

TUDP = class(TOleControl)
  private
    FOnError: TUDPError;
    FOnDataArrival: TUDPDataArrival;
  protected
    procedure InitControlData; override;
  public
    procedure AboutBox; stdcall;
    procedure SendData(const data: Variant); stdcall;
    procedure GetData(var data: Variant; const type_: Variant); stdcall;
    property RemoteHostIP: string index 1001 read GetStringProp;
    property LocalHostName: string index 1002 read GetStringProp;
    property LocalIP: string index 1003 read GetStringProp;
    property SocketHandle: Integer index 1004 read GetIntegerProp;
  published
    property RemoteHost: string index 0 read GetStringProp write
        SetStringProp stored False;
    property RemotePort: Integer index 502 read GetIntegerProp write
        SetIntegerProp stored False;
    property LocalPort: Integer index 1010 read GetIntegerProp write
        SetIntegerProp stored False;
    property OnError: TUDPError read FOnError write FOnError;
    property OnDataArrival: TUDPDataArrival read FOnDataArrival write
        FOnDataArrival;
  end;
```

LES ACTIVEX

PRÉSENTATION

Microsoft a d'abord lancé OLE (*Object Linking and Embedding*, ou liaison et incorporation d'objets) comme standard permettant à des objets de communiquer avec une application hôte. La spécification initiale avait pour but de permettre à une application telle qu'Excel d'incorporer une feuille de calcul dans n'importe quelle autre application prenant en charge le standard OLE. OLE 1.x était dépourvu de certaines fonctionnalités indispensables, si bien qu'une spécification OLE 2.0 a été mis en place, puis implémentée. OLE représentait les fondations d'une technologie permettant le partage d'objets génériques. Cette technologie s'est appelée COM (*Component Object Model* ou Modèle objet de composant) et a été utilisée pour créer une spécification pour les composants OCX. L'acronyme OLE s'est révélé réducteur puisque COM était loin de se cantonner à l'incorporation et à la liaison d'objets. Microsoft a alors développé le standard ActiveX actuel, ainsi que les composants ActiveX, qui sont les successeurs d'OLE et des OCX fondés sur l'architecture COM.

On peut se représenter COM comme le standard binaire de partage de composants entre deux morceaux de code. COM permet de séparer l'implémentation d'un objet des fonctions que cet objet effectue. Les fonctions qu'il effectue sont décrites dans ses interfaces. Une interface est une méthode d'accès à un ensemble de fonctions logiquement apparentées, que peut implémenter un objet. Chaque classe d'objet possède un identificateur (ID) de classe unique (CLSID) qui prend en charge un ensemble arbitraire d'interfaces. Toutes les classes doivent prendre en charge l'interface IUnknown qui peut être ensuite utilisée pour accéder aux interfaces qu'elles gèrent. Ceci s'effectue par le biais de la fonction QueryInterface, qui est toujours fournie dans l'interface IUnknown. Celle-ci permet à une application de demander à un objet s'il prend en charge les fonctions lui permettant d'effectuer telle ou telle tâche. L'objet répond alors par oui ou par non. Ce modèle objet est très puissant car il permet à une application de déterminer cela en phase d'exécution.

Un objet COM est implémenté par le biais de plusieurs méthodes. Il peut être compilé en une DLL ou un OCX s'exécutant dans le même espace de processus que l'application qui l'appelle. Il peut également être lancé dans son propre espace, sous forme d'exécutable compilé. Avec COM distribué (DCOM), l'objet peut s'exécuter sur une machine différente, n'importe où dans le monde. Les services système COM simplifient l'appel d'objets COM, même si le code d'implémentation se trouve dans un processus ou sur une machine différente.

Les composants ActiveX sont des objets COM qui implémentent un ensemble d'interfaces de base permettant au composant d'être incorporé à des applications qui accueillent des composants ActiveX. Avec Delphi, il est très simple de créer des composants ActiveX, même à partir d'un composant visuel Delphi déjà existant. L'application hôte peut alors manipuler les propriétés et répondre aux événements tout comme une application Delphi avec des composants visuels. Vous pouvez également ajouter de nouveaux événements, propriétés et méthodes au composant ActiveX pour lui donner des fonctionnalités supplémentaires.

CRÉATION DE CONTRÔLES ACTIVEX

La première étape pour créer un composant ActiveX consiste à créer une nouvelle bibliothèque ActiveX. Pour la créer, choisissez Fichier, Nouveau dans le menu, sélectionnez l'onglet ActiveX, puis choisissez Bibliothèque ActiveX. Vous créez ainsi un nouveau projet qui se compilera sous forme de fichier .OCX (le module qui stocke les composants ActiveX). Ensuite, choisissez Fichier, Nouveau dans le menu Delphi. Dans la boîte de dialogue qui apparaît alors, choisissez Contrôle ActiveX. L'assistant du même nom apparaît, c'est celui qui générera le code nécessaire à la création d'un contrôle ActiveX à partir d'un composant visuel déjà existant.

L'assistant a besoin de trois informations : le composant visuel sur lequel sera fondé le composant ActiveX, la classe du nouveau composant ActiveX et l'emplacement du futur fichier d'implémentation. D'autres options vous permettent d'utiliser des licences de conception, le contrôle des versions et une boîte A propos. Pour notre exemple, nous allons partir du composant visuel bouton horloge (TButtonClock) pour en faire un contrôle ActiveX.

L'Assistant génère tout le code nécessaire à la compilation du composant en un composant ActiveX. Pour compiler le contrôle, il suffit de choisir Projet, Compiler dans le menu Delphi. Pour ajouter des fonctionnalités à un contrôle ActiveX, vous pouvez employer deux méthodes.

La première consiste à les ajouter au composant visuel sur lequel est basé le contrôle ActiveX et à le construire à nouveau. L'autre méthode consiste à ajouter directement les fonctionnalités au composant ActiveX. Le code source généré par l'Assistant contrôle ActiveX figure ci-dessous.

Le code source du contrôle ActiveX généré par l'assistant contrôle ActiveX à partir du composant bouton horloge

```
unit ActXClockImpl;
interface
uses
    Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms,
    [ccc]StdCtrls,ComServ, StdVCL, AXCtrls, ActXClockPR_TLB, unitTBC;
type
    TActXClockX = class(TActiveXControl, IActXClockX)
private
    { Déclarations privées }
    FDelphiControl: TButClock;
    FEvents: IActXClockXEvents;
protected
    { Déclarations protégées }
    procedure InitializeControl; override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure DefinePropertyPages(DefinePropertyPage:
    [ccc]TDefinePropertyPage); override;
    function Get_Cancel: WordBool; safecall;
    function Get_Caption: WideString; safecall;
    function Get_Cursor: Smallint; safecall;
```

```
        function Get_Default: WordBool; safecall;
        function Get_DragCursor: Smallint; safecall;
        function Get_DragMode: TxDragMode; safecall;
        function Get_Enabled: WordBool; safecall;
        function Get_Font: Font; safecall;
        function Get_ModalResult: Integer; safecall;
        function Get_Visible: WordBool; safecall;
        procedure Click; safecall;
        procedure Set_Cancel(Value: WordBool); safecall;
        procedure Set_Caption(const Value: WideString); safecall;
        procedure Set_Cursor(Value: Smallint); safecall;
        procedure Set_Default(Value: WordBool); safecall;
        procedure Set_DragCursor(Value: Smallint); safecall;
        procedure Set_DragMode(Value: TxDragMode); safecall;
        procedure Set_Enabled(Value: WordBool); safecall;
        procedure Set_Font(const Value: Font); safecall;
        procedure Set_ModalResult(Value: Integer); safecall;
        procedure Set_Visible(Value: WordBool); safecall;
end;

implementation
{ TActXClockX }

procedure TActXClockX.InitializeControl;
begin
        FDelphiControl := Control as TButClock;
end;

procedure TActXClockX.EventSinkChanged(const EventSink: IUnknown);
begin
        FEvents := EventSink as IActXClockXEvents;
end;

procedure TActXClockX.DefinePropertyPages(DefinePropertyPage:
[ccc]TDefinePropertyPage);
begin
        { Définissez les pages de propriété ici. Celle(s)-ci sont définies en
        appelant DefinePropertyPage avec l'id de classe de la page. Par exemple,
        DefinePropertyPage(Class_ActXClockXPage); }
end;

function TActXClockX.Get_Cancel: WordBool;
begin
        Result := FDelphiControl.Cancel;
end;

function TActXClockX.Get_Caption: WideString;
begin
        Result := WideString(FDelphiControl.Caption);
end;
```

```
function TActXClockX.Get_Cursor: Smallint;
begin
        Result := Smallint(FDelphiControl.Cursor);
end;

function TActXClockX.Get_Default: WordBool;
begin
        Result := FDelphiControl.Default;
end;

function TActXClockX.Get_DragCursor: Smallint;
begin
        Result := Smallint(FDelphiControl.DragCursor);
end;

function TActXClockX.Get_DragMode: TxDragMode;
begin
        Result := Ord(FDelphiControl.DragMode);
end;

function TActXClockX.Get_Enabled: WordBool;
begin
        Result := FDelphiControl.Enabled;
end;

function TActXClockX.Get_Font: Font;
begin
        GetOleFont(FDelphiControl.Font, Result);
end;

function TActXClockX.Get_ModalResult: Integer;
begin
        Result := Integer(FDelphiControl.ModalResult);
end;

function TActXClockX.Get_Visible: WordBool;
begin
        Result := FDelphiControl.Visible;
end;

procedure TActXClockX.Click;
begin
end;

procedure TActXClockX.Set_Cancel(Value: WordBool);
begin
        FDelphiControl.Cancel := Value;
end;
```

```
procedure TActXClockX.Set_Caption(const Value: WideString);
begin
        FDelphiControl.Caption := TCaption(Value);
end;

procedure TActXClockX.Set_Cursor(Value: Smallint);
begin
        FDelphiControl.Cursor := TCursor(Value);
end;

procedure TActXClockX.Set_Default(Value: WordBool);
begin
        FDelphiControl.Default := Value;
end;

procedure TActXClockX.Set_DragCursor(Value: Smallint);
begin
        FDelphiControl.DragCursor := TCursor(Value);
end;

procedure TActXClockX.Set_DragMode(Value: TxDragMode);
begin
        FDelphiControl.DragMode := TDragMode(Value);
end;

procedure TActXClockX.Set_Enabled(Value: WordBool);
begin
        FDelphiControl.Enabled := Value;
end;

procedure TActXClockX.Set_Font(const Value: Font);
begin
        SetOleFont(FDelphiControl.Font, Value);
end;

procedure TActXClockX.Set_ModalResult(Value: Integer);
begin
        FDelphiControl.ModalResult := TModalResult(Value);
end;

procedure TActXClockX.Set_Visible(Value: WordBool);
begin
        FDelphiControl.Visible := Value;
end;

initialization
        TActiveXControlFactory.Create( ComServer, TActXClockX, TButClock,
                Class_ActXClockX, 1, '', 0);
end.
```

*Une nouvelle classe est générée (*TActXClockX*). Elle contient un composant visuel* TbutClock *dans la section* private *de la définition de classe. Toutes les propriétés et méthodes du composant ActiveX sont définies comme des procédures et des fonctions dans sa déclaration.Ainsi, la propriété* Cursor *est implémentée avec la fonction* Get_Cursor *et la procédure* Set_Cursor. *Ces procédures sont appelées lorsque* Cursor *est définie ou lue. Leur implémentation est automatiquement générée par l'Assistant contrôle ActiveX.*

En plus du fichier d'implémentation ActiveX, l'Assistant Control construit une bibliothèque de types, qui définit les interfaces et les propriétés du composant dans une bibliothèque ActiveX.

Delphi propose un éditeur de bibliothèque de types vous permettant de modifier (et de consulter) les informations que celle-ci contient sur un contrôle ActiveX. Pour consulter cette bibliothèque, choisissez Voir, Bibliothèque de types, et vous pourrez alors voir quels contrôles, interfaces et pages de propriétés se trouvent dans le projet, ainsi que leurs propriétés, événements et méthodes.

AJOUTER DIRECTEMENT UNE MÉTHODE

Il est également très facile d'ajouter des propriétés, des événements et des méthodes directement dans un contrôle ActiveX. Pour cela, nous allons ajouter une nouvelle méthode, MakeBold, qui fera passer en gras le texte du libellé. Pour l'ajouter, choisissez Editer, Ajouter à l'interface dans le menu Delphi. La boîte de dialogue Ajout à l'interface apparaît alors. Assurez-vous que Interface est définie comme Propriétés/méthodes, et entrez procedure MakeBold; pour la déclaration .

Vous avez ainsi effectué trois tâches : la méthode MakeBold a été ajoutée à la définition d'interface dans la bibliothèque de types et à la définition de classe, et un squelette de la procédure MakeBold a été créé. Le voici :

```
procedure TActXClockX.MakeBold;
begin
end;
```

Il vous incombe alors de compléter ce code en ajoutant celui qui modifie la police. La procédure finale est la suivante :

```
procedure TActXClockX.MakeBold;
begin
FDelphiControl.Font.Style := [fsBold];
end;
```

Le FDelphiControl référencé dans cette procédure est une instance du composant TButClock, qui est encapsulé dans le contrôle ActiveX. Lorsque la méthode MakeBold est appelée sur le composant ActiveX TActXClockX, la procédure affecte fsBold à la propriété Font.Style dans le composant encapsulé. Vous pouvez voir la déclaration de FDelphiControl dans la définition de classe.

Lorsque vous compilez le projet, vous obtenez un OCX contenant l'implémentation du composant ActiveX. Les composants ActiveX doivent être enregistrés dans un système avant utilisation. La bibliothèque ActiveX s'autoenregistre si l'application hôte peut appeler la procédure d'enregistrement. Il est également possible d'enregistrer le composant à partir de l'EDI Delphi en choisissant Exécuter, Recenser Serveur ActiveX. Ici, sauvegardez-le immédiatement à l'aide de cette option de menu.

Vous pouvez également utiliser Dé-recenser serveur ActiveX pour désinstaller un contrôle de votre machine. Le moyen le plus simple pour le tester consiste à utiliser la commande Déploiement Web pour que Delphi génère une page Web de test. Nous allons compliquer un peu ce principe dans la partie suivante en incorporant le composant dans une page Web contenant un script qui permettra à ce composant d'interagir avec d'autres composants de la page.

DÉPLOIEMENT D'ACTIVEX SUR LE WEB

L'aspect le plus séduisant des ActiveX est qu'ils sont indépendants du langage et de l'application. Dans l'exemple qui va suivre, nous allons créer une page Web en comportant deux : le désormais célèbre bouton horloge et le bouton poussoir Microsoft standard. On ajoutera du VBScript à la page pour que, lorsque l'utilisateur clique sur le bouton poussoir, la police du bouton horloge passe en gras. Le code HTML correspondant à cette page figure ci-dessous.

Code HTML pour créer une page contenant des composants ActiveX

```
<HTML>
<HEAD>
<TITLE>Des composants ActiveX marchant main dans la main ! </TITLE>
</HEAD>
<BODY>
<H2> Des composants ActiveX marchant main dans la main !</H2>
<HR>
<CENTER><Table Border=1>
<TR><TD ALIGN=CENTER>
<OBJECT ID="PushForBold" WIDTH=203 HEIGHT=32
        CLASSID="CLSID:D7053240-CE69-11CD-A777-00DD01143C57">
        <PARAM NAME="VariousPropertyBits" VALUE="268435483">
        <PARAM NAME="Caption" VALUE="Cliquez  pour enrichir l'horloge ">
        <PARAM NAME="Size" VALUE="4313;678">
</OBJECT>
</TD></TR>
<TR><TH>Appuyez sur le bouton ci-avant pour faire passer l'horloge en gras
<TH></TR>
<TR><TD ALIGN=CENTER>
<OBJECT ID="ClockButton" WIDTH=75 HEIGHT=25
        CLASSID="CLSID:C8EE0B43-8C8F-11D0-9FB3-444553540000">
        </OBJECT>
```

```
</TD></TR></TABLE></CENTER><HR>
L'horloge et le bouton sont des composants ActiveX. Le reste est du HTML on ne
peut plus classique

<SCRIPT LANGUAGE="VBScript">
        Sub PushForBold_Click()
                call ClockButton.MakeBold()
        end sub
</SCRIPT>
</BODY>
</HTML>
```

Chacun des composants ActiveX est marqué par une balise <OBJECT> *qui inclut un ID le référençant dans la page. Dans cette balise, on spécifie des informations concernant le composant. La plus cruciale se trouve dans la propriété* CLASSID. *C'est là qu'on précise quel composant ActiveX sera placé sur la page pour cet objet. L'ID de classe correspond au* CoClass GUID *affiché dans la bibliothèque de types de Delphi. Les autres paramètres de la balise* <OBJECT> *spécifient d'éventuels paramètres d'initialisation pour l'objet. Ainsi, sur le bouton de commande, on saisit le libellé* Cliquez ici pour enrichir l'horloge. *Il faut aussi spécifier un Gestionnaire d'événements en faisant en sorte que le VBScript appelle la méthode* MakeBold *lorsqu'on appuie sur le bouton poussoir. Vous auriez pu tout aussi bien utiliser JavaScript ou d'autres langages de script.*

LES COMMUNICATIONS ENTRE APPLICATIONS WINDOWS

Il est de plus en plus fréquent de chercher à faire communiquer des applications (développées la plupart du temps avec des produits différents) entre elles.

On parle fréquemment de modèle "client - serveur" lorsqu'il s'agit de créer deux applications (une application cliente et une application serveur) susceptibles d'entretenir un dialogue entre elles : l'application cliente effectue une requête auprès du serveur pour obtenir un "service". Celui-ci en retour renvoie au client le service demandé (demande d'information ou mise à jour d'une donnée gérée par le serveur).

Mais si ce mode de fonctionnement est intéressant dans les cas les plus ambitieux (application basée sur un serveur SQL en particulier), mettant en oeuvre plusieurs machines au sein d'un réseau, on ne souhaite souvent que faire communiquer deux applications travaillant sur une même station.

A ce niveau les possibilités de communications sont les suivantes :

- L'utilisation du presse-papiers ;
- La mise en place de liens DDE ou OLE ;

Selon les besoins on pourra mettre en oeuvre une ou plusieurs de ces méthodes. Il est du ressort du concepteur du système d'information à mettre en oeuvre de bien déterminer ce dont il a réellement besoin tant les conséquences sur la programmation peuvent être importantes.

GESTION DU PRESSE-PAPIERS

Le presse-papiers (Clipboard en anglais) est un moyen simple proposé par Windows pour permettre le transfert de données (texte ou images) entre applications.

Lorsque l'on copie (ou que l'on coupe) une donnée quelconque à partir d'une application vers le presse-papiers, toute application Windows active (y compris l'application émettrice de la donnée) peut récupérer cette donnée à l'aide d'une opération de "collage".

On accède au presse-papiers à l'aide :

- Du menu **EDITION** présent dans la plupart des applications.
- Par des combinaisons de touches standardisées Ctrl + C : Copier,
  Ctrl + X : Couper,
  Ctrl + V : Coller.

Le fait de déposer une donnée dans le presse-papiers écrase le contenu précédent. Par contre le contenu du presse-papiers est accessible tant que l'on n'est pas sorti de Windows.

DELPHI propose une classe spécifique appelée TClipBoard, invisible, pour gérer les relations avec le presse-papiers de Windows. Elle déclare automatiquement un objet à ce type, appelé ClipBoard, pour chaque projet ouvert.

Comme l'objet est invisible, tout comme les objets Application et Screen, on ne peut pas visualiser ses propriétés et ses événements associés dans l'inspecteur de propriétés.

Rajouter l'unité ClipBrd dans la clause 'uses' de la feuille.

Manipulations directes du presse-papiers :

Pour insérer puis récupérer des données dans le presse-papiers on doit invoquer les propriétés suivantes :

- **AsText** : Pour des données de type texte ;
- **Assign** : Pour une image.

Les méthodes **GetComponent ()** et **SetComponent()** sont utilisées pour copier puis coller un composant.

Exemples :

```
Clipboard.AsText := Edit.Text ;
Label1.Caption := Clipboard.AsText ;
{ Copie le contenu de la zone d'édition puis le renvoie comme texte de l'étiquette }
```

```
Clipboard.Assign  (Image1 ) ;
Image2.Assign (ClipBoard ) ;
{ Copie Image1, qui doit être un objet graphique, puis le colle dans un autre objet graphique.}
```

La méthode Clear efface le contenu du presse-papiers.

Il est par ailleurs possible de connaître les formats graphiques gérés en invoquant la méthode Formats (ces formats sont des entiers dont il faut connaître l'équivalence).

Pour éviter qu'une autre application vienne écraser le contenu du presse-papiers, on utilise la méthode Open qui préserve le contenu (le presse-papiers aura alors en mémoire plusieurs objets).

Il faut fermer le presse-papiers (méthode Close) lorsqu'il a été ouvert avec la méthode Open.

Accès à partir d'autres composants :
Plusieurs composants possèdent des méthodes qui permettent d'accéder directement au presse-papiers sans avoir à utiliser la variable globale ClipBoard.

Les différents composants permettant le saisie de texte (TEdit, TMemo) possèdent les méthodes d'accès suivantes :

- **CutToClipBoard** ;
- **CopyToClipBoard** ;
- **PasteFromClipBoard** ;

```
procedure TForm1.Button1Click(Sender: TObject);
begin
        Memo1.CopyToClipboard;
        Edit1.PasteFromClipboard;
end;
```

La partie du texte sélectionnée préalablement dans Memo est recopiée, via le presse-papiers, dans la zone de saisie

UTILISATION DE DDE

GÉNÉRALITÉS SUR LE PROTOCOLE DDE :

Le protocole DDE (Dynamic Data Exchange) est un protocole mis au point par Microsoft, dans l'environnement Windows, pour que deux applications soient en mesure d'échanger des données. L'application qui fournit les données est appelée 'serveur', l'application qui les reçoit est appelée 'client' (mais il peut y avoir des liens établis qui permettent la communication dans les deux sens).

Actuellement la plupart des applications tournant sous Windows sont susceptibles d'être 'client DDE'. Mais peu sont conçues pour être 'serveur DDE'.

On appelle 'service' l'identifiant du serveur DDE. Ce peut être le nom de l'exécutable de l'application (cas général) mais ce n'est pas une obligation.

COMPOSANTS DELPHI POUR CONSTITUER UN LIEN DDE :

DELPHI est fourni avec 4 composants, contenus dans l'onglet 'Système' de la palette de composants, qui permettent la constitution d'un lien DDE. Ils sont utilisés différemment selon les besoins.
Il y a lieu de distinguer les composants chargés d'assurer la liaison entre le client et le serveur et ceux chargés du transfert de données proprement dit.

Tous ces composants sont des composants invisibles.

Composant	Utilisation
DDEServerConv	Permet l'établissement d'une conversation d'un serveur DDE avec un client DDE
DDEServerItem	Permet d'envoyer des données au client
DDEClientConv	Permet l'établissement d'une conversation DDE entre un client et un serveur DDE
DDEClientItem	Permet au client de récupérer une donnée sur un serveur DDE

CRÉATION D'UN LIEN DDE ENTRE UNE APPLICATION EXISTANTE ET UNE APPLICATION DELPHI.

Préalable :
Pour créer un lien DDE entre une application existante et une application DELPHI il faut d'abord s'assurer, en lisant la documentation, que l'application concernée a bien été conçue comme 'serveur DDE'. Aujourd'hui la plupart des applications de bureautique majeures sont 'serveur DDE' mais ce n'est pas encore toujours le cas.

Une fois cette vérification effectuée, il faut créer le lien DDE entre l'application DELPHI en cours de réalisation et l'application serveur.

Comme on se trouve dans le cas où s'est le 'client' qui va chercher des données dans l'application 'serveur', les composants à utiliser sont : DDEClientConv et DDEClientItem.

> Pour comprendre la méthode de création d'un lien DDE, on utilisera le tableur Excel et le fichier BUDGET.XLS se trouvant normalement dans le sous-répertoire EXCELCBT.

Configuration des composants DDEClientConv et DDEClientItem :
Le composant DDEClientConv sert à créer et à maintenir le lien DDE entre l'application cliente et l'application serveur. Les propriétés à configurer sont les suivantes :

Propriété	Action
ConnectMode	Selon la valeur (ddeAutomatic / ddeManual) le lien DDE sera réalisé automatiquement à la création de la feuille contenant le composant ou sur action programmée.
DdeService	Indique le nom de l'application serveur.
DdeTopic	Indique le chemin d'accès au fichier de l'application serveur dans lequel se trouvent les données à transférer (cas général).
ServiceApplication	Si une valeur est spécifiée (elle correspond alors -dans le cas général- à la valeur de DDeService) l'application serveur est chargée en mémoire et exécutée pour pouvoir réaliser le lien DDE.

Le composant DDEClientItem sert à distribuer les données du serveur dans l'application cliente. S'il n'y a qu'un seul composant DDEClientConv (pour un lien à constituer avec un serveur DDE), il y a autant de composants DDEClientItems que de données à transférer.

Les propriétés à configurer sont les suivantes :

Propriété	Action
DdeConv	Indique le nom du composant DDEClientConv réalisant le lien DDE
DdeItem	Indique le nom de l'élément source de la donnée (il peut y avoir plusieurs éléments).
Lines	Contient la donnée à transférer au format texte (utile quand il y a plusieurs données)
Text	Contient la donnée à transférer au format texte

La réalisation d'un lien DDE à l'exécution ne peut se faire que si les deux composants ont, au préalable été correctement initialisés lors de la phase de conception. En effet certaines propriétés (DdeService, DdeTopic de DDEClientconv et DdeItem de DDEClientItem) doivent être initialisées avec des valeurs utilisant la syntaxe propre à chaque serveur DDE. DELPHI propose une méthode pour initialiser correctement certaines de ces valeurs lors de la conception.

Configuration des composants lors de la phase de conception :

Pour configurer, lors de la phase de conception, les deux composants une méthode simple consiste à réaliser les actions suivantes :

1. Lancer l'application serveur et charger le fichier contenant les données à transférer.

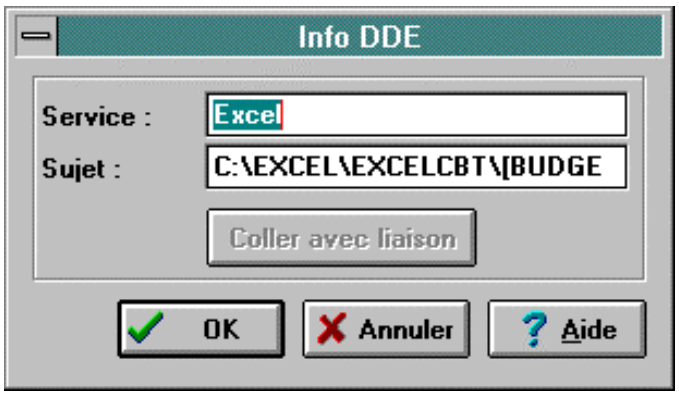
2. Sélectionner une cellule du tableau (ce peut être n'importe laquelle mais si le lien DDE n'est réalisé que sur une seule cellule autant sélectionner celle-ci).

3. Copier son contenu dans le presse-papiers.

4. Revenir dans DELPHI. Déposer un composant DDEClientConv sur la fiche adéquate du projet.

5. Double-cliquer sur la propriété DdeService ou DdeTopic (l'une ou l'autre). Une boite de dialogue, appelée InfoDDE, s'affiche alors. Le fait de cliquer sur le bouton 'Coller avec liaison' renseigne automatiquement les deux zones d'édition (nom du service et chemin d'accès à ce service) :

6. Valider le choix. Les propriétés du composant sont automatiquement renseignées.



Utilisation de la boite de dialogue InfoDDE configurant automatiquement le composant DDEClientConv.

En sélectionnant le composant DDEClientServ on indique le nom du composant DDEClientConv dans la liste proposée par la propriété DdeConv.

En sélectionnant la bonne donnée dans la propriété DdeItem les propriétés Lines et Text se renseignent automatiquement.

Si l'on modifie à la main la valeur de la propriété DdeItem, les valeurs des propriétés Lines et Text sont modifiées automatiquement.

A ce niveau, il est possible d'exécuter l'application : comme le serveur DDE est chargé en mémoire, le lien DDE est réalisé. Il suffit de créer un gestionnaire d'événement associé à l'événement OnCreate de la feuille pour récupérer dans les composants DELPHI adéquats les valeurs ainsi transférées (un composant TEdit par exemple). Dans la plupart des cas il faut penser à réaliser une conversion de type (cas de valeurs numériques transférées en tant que chaînes de caractères).

Exemple :

Lancer Excel et charger le fichier BUDGET.XLS. Sélectionner une cellule. La copier dans le presse-papiers. Revenir sous DELPHI. Cliquer dans la propriété DdeService (ou DdeTopic) du composant DDEClientConv. La boite de dialogue de configuration apparaît. Cliquer sur le bouton 'Coller avec liaison' et valider le choix.

Sélectionner le composant DDEClientItem. Connecter le, via la propriété DdeConv, au composant DDEClientConv. Vérifier que les renseignements contenus dans les autres propriétés correspondent bien à ceux de la feuille BUDGET.XLS.

Déposer un composant TEdit et programmer l'événement OnCreate de la feuille comme suit :

TEdit.Text := DdeClientItem.Text ;

Exécuter le programme.

Programmation de la réalisation du lien DDE à l'exécution :

Une fois les composants initialisés il faut créer le code permettant de réaliser le lien DDE à l'exécution.

Deux modes de connexion sont possibles, en fonction de la valeur affectée à la propriété ConnectMode du composant DDEClientConv. Le mode automatique est le plus simple à réaliser.

Pour réaliser le lien il faut :

- Invoquer la méthode Setlink() du composant DDEClientConv afin d'initialiser à l'exécution les propriétés DdeService et DdeTopic.

  Il faut absolument utiliser cette méthode car il n'est pas possible d'initialiser directement les deux propriétés. Par contre il faut récupérer les valeurs d'initialisation exactes fournies lors de la phase de conception.

- Ensuite initialiser la propriété DdeItem du composant DDEClientItem avec l'identifiant correct de la donnée à transférer.

Si on utilise l'exemple précédent le code correspondant est le suivant :

```
procedure TF_DDE.FormCreate (Sender: TObject);
begin
        Screen.Cursor := crDefault ;
        with DDEClientConv1 do
        begin
                if SetLink('Excel', 'C:\DELPHI\ESSAIS\DDE \
                [BALANCE.XLS]Feuil1')
                {la syntaxe est donnée par la phase de conception }
                then
```

```
                DdeClientItem1.DdeItem := 'L6C4' { Idem }
            else
                MessageDlg ( 'Lien non établi', mtInformation,[mbOK], 0);
        end ;
end ;

 procedure TF_DDE.DdeClientItem1Change(Sender: TObject);
begin
        Edit1.Text := DdeClientItem1.Text ;
end;
```

L'événement OnChange n'est activé que si la connexion est réalisée

A ce niveau apparaît alors un gros problème qui peut empêcher la bonne réalisation du lien DDE. Si l'application appelée est longue à se charger, l'appel à **SETLINK** sera réalisé avant qu'elle soit réellement en place et le lien ne sera alors pas établi.

Dans l'exemple précédent, le chargement d'Excel puis de la feuille BALANCE.XLS prend trop de temps et n'est pas réalisé avant l'appel de la méthode **SETLINK**.

Plusieurs solutions sont alors à la disposition du programmeur :

- Si le lien DDE doit être souvent utilisé tout au long de l'application, le serveur DDE peut être chargé dès le démarrage (même si le lien n'est réalisé qu'ensuite). Cette solution risque d'être pénalisante en terme de ressources système utilisées.

  Pour lancer l'application serveur il faudra utiliser la fonction WinExec () de l'API Windows.

- On peut aussi charger le serveur DDE juste avant de créer la feuille contenant le composant DDEClientConv. Cette solution permet de laisser un peu plus de temps avant l'appel de la méthode Setlink (). Mais cela n'est pas toujours suffisant.

On constate donc qu'il n'y a pas de solution réellement satisfaisante. Le temps de chargement du programme serveur dépendant de nombreux paramètres (vitesse du processeur, taille de la mémoire, charge du système à l'instant du chargement, etc...), il n'est pas possible de faire des à priori faisant intervenir des temporisations quelconques dans l'algorithme d'exécution.

Le code correspondant est alors le suivant :

```
procedure TForm1.FormCreate ( Sender: TObject ) ;
var
        ExcelDir : array [ 0 .. 145 ] of char ;
        Retour : Integer ;
begin
        Screen.Cursor := crHourGlass ;
        StrPCopy ( ExcelDir , 'c:\excel\excel.exe c:\delphi\essais\dde\balance.xls' ) ;
        { Transformation d'une chaîne Pascal en chaîne AZT }
```

```
        Retour := WinExec ( ExcelDir , SW_MINIMIZE ) ;
        { Lancement du serveur }
        case Retour of                  { Test d'erreurs éventuelles }
                2 ,3 : Messagedlg ( 'Excel non trouvé !!!', mtError ,[mbOk] , 0 ) ;
                8 : Messagedlg ( 'Mémoire insuffisante !!!',mtError , [mbOk] , 0 ) ;
         end ;
end;
procedure TForm1.Etablirlelien1Click(Sender: TObject);
{ Gestionnaire d'événement lancé par clic sur un item de menu 'Etablir le lien' }
begin
        with DDEClientConv1 do
        begin
                if                              SetLink('Excel','C:\DELPHI\ESSAIS\DDE\
[BALANCE.XLS]Feuil1')           then begin
                Screen.Cursor := crHourGlass ; {Fait patienter l'utilisateur}
                DdeClientItem1.DdeItem := 'L6C4' ;
                Edit1.Visible := True ;         { Edit1 était invisible au départ }
                end
                else
                MessageDlg('Lien non établi', mtInformation,[mbOK], 0);
        end ;
end;

procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
        Edit1.Text := DdeClientItem1.Text ;
end;
{ Transfert de la donnée lorsque le lien est établi }

procedure TForm1.Edit1Change(Sender: TObject);
begin
        Screen.Cursor := crDefault ;
end;
```

Fermeture d'un lien DDE :

Pour fermer un lien DDE il suffit d'invoquer la méthode CloseLink du composant DDEClientConv.

Cette fermeture ne décharge pas l'application serveur de la mémoire.

TRANSFERT DE DONNÉE D'UN CLIENT DDE VERS LE SERVEUR

Dans certains cas il est possible que l'on souhaite transférer une donnée à partir du client vers le serveur.

Par exemple on peut imaginer l'envoi d'une donnée dans une feuille de calcul EXCEL. Cette nouvelle donnée modifie les valeurs contenues dans la feuille. Il suffit alors de récupérer une

valeur résultante selon un lien DDE utilisé, dans le sens serveur - client , comme définit précédemment.

Pour ce faire, il faut utiliser un deuxième composant DDEClientItem, qui pourra être connecté au même composant DDEClientConv que celui utilisé pour établir le lien serveur-client.

Le transfert de la donnée se fait comme suit :

- La propriété DDEItem, définissant la donnée objet du lien, est initialisée comme précédemment après un appel à la méthode Setlink ().

- En programmation il faut invoquer la méthode PokeData () du composant DDEclientConv qui spécifie la valeur de la donnée à transférer.

On a alors le code suivant :

```
procedure TForm1.Envoidunedonne1Click(Sender: TObject);
var
        donnee : PChar ;

begin
        StrPCopy ( donnee , Edit2.Text ) ;
        { Transformation d'une chaine Pascal en chaine AZT }
        if not DDEClientConv1.PokeData(DDEClientItem2.DDEItem, donnee ) then
                MessageDlg ( 'Envoi de la donnée refusé ', mtError ,[mbOK] , 0 ) ;
end;
```

La méthode PokeData () renvoie un booléen qui est à true si le transfert a été réalisé

Il y a toutefois à ce niveau là un problème majeur : la donnée transférée l'est sous forme d'une chaîne. Si l'emplacement où elle doit être transférée sur le serveur n'accepte pas ce format (exemple : une cellule d'Excel)... ça plante. Au lieu d'envoyer directement la donnée il faut donc transmettre - via la méthode ExecuteMacro() - une macro (écrite selon la syntaxe du programme serveur) permettant la conversion d'une donnée texte en donnée numérique.

CRÉATION D'UN SERVEUR DDE

DELPHI fournit les moyens de constituer un serveur DDE.

Pour cela il suffit de créer une application quelconque et utiliser les composants DDEServConv et DDEServItem disponibles dans la palette de composants 'Système'.

Une fois ces deux composants déposés et liés entre eux (propriété ServConv du composant DDEServItem) l'application est considérée comme un serveur DDE et peut donc être liée à une application cliente selon les mécanismes vus précédemment.

L'utilisation d'un serveur DDE créé spécialement n'est pas très courante. Dans la plupart des cas on crée un lien DDE parce que l'on doit utiliser les capacités spécifiques d'une application commerciale.

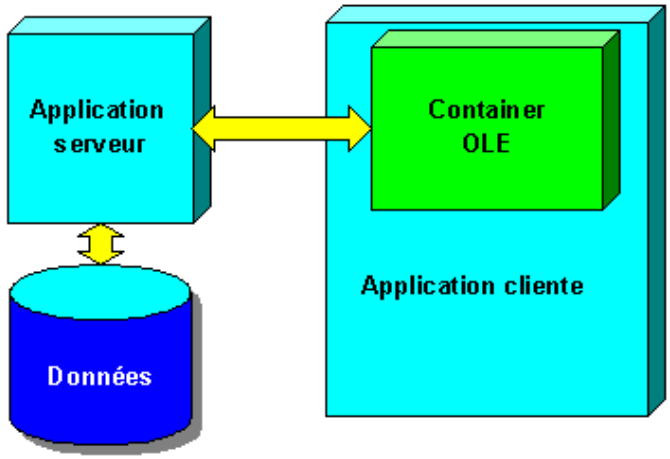
UTILISATION DE OLE

PRINCIPES ET GÉNÉRALITÉS

Le protocole OLE (Object Linking and Embedding), qui en est à sa version OLE 2.0, est une évolution du protocole DDE et permet, comme ce dernier, le partage de données (appelées Objets) entre applications. A terme il est amené à se substituer à DDE bien qu'il ne propose pas tout à fait les mêmes services.

Cependant contrairement à DDE, lorsqu'on utilise OLE on accède directement à l'application serveur et on travaille dans son sein (en particulier les données manipulées sont gérées et stockées par l'application serveur). Il faut ensuite fermer l'application serveur pour revenir dans l'application cliente.

Si l'on souhaite ramener une donnée dans l'application cliente, il faut penser à créer un lien DDE entre les deux applications pour que celle-ci puisse être transmise.



Un container OLE n'est qu'une visualisation de l'application Serveur OLE au sein de l'application cliente. Les données restent gérées par l'application serveur.

Un objet OLE peut être de tout type (données, image, dessin, texte,....). Il est affiché dans une fenêtre de l'application cliente (dite 'application conteneur'). Un double-clic sur l'objet appelle l'application serveur pour modification éventuelle.

Il y a deux manières d'inclure un objet OLE dans une application conteneur :

- Un objet peut être stocké dans un fichier externe. Il peut alors être partagé par plusieurs clients conteneurs (et par le serveur). Chaque modification est prise en compte par tous les clients. Dans ce cas on dit que l'objet est **lié**.

- Un objet peut être stocké dans un fichier géré par une application conteneur (soit de manière externe soit au sein de l'exécutable). Seule cette dernière y a accès. Dans ce cas l'objet est dit **imbriqué**.

    Les objets liés sont stockés dans des fichiers. On ne peut lier un objet OLE que s'il a été au préalable créé au sein de l'application serveur.

Les objets incorporés sont stockés dans l'application conteneur. Il est possible de créer un objet OLE à partir de l'application conteneur. Mais pour que les données modifiées soient accessibles d'une exécution à l'autre il faut qu'elles soient stockées dans un fichier.

LE COMPOSANT OLECONTAINER

DELPHI dispose du composant OLEContainer pour permettre la création d'un lien OLE entre une application conteneur et un serveur OLE. Ce composant simplifie énormément la mise en oeuvre d'un lien OLE en encapsulant entièrement le mécanisme de l'API Windows.

Ses principales propriétés sont :

Propriété	Action
AutoSize	Permet d'ajuster la taille du conteneur OLE à celle de l'objet OLE qu'il contient.
Modified	Indique si l'objet OLE a été modifié depuis son initialisation.
ObjClass **ObjDoc** **ObjItem**	Paramètres d'initialisation permettant de désigner l'objet OLE (type, emplacement, etc..)

Le composant TOleContainer est un composant visible que l'on peut redimensionner à la conception.

CRÉATION D'UNE APPLICATION CONTENEUR OLE

Pour créer une application conteneur il suffit d'utiliser un composant TOleContainer et initialiser celui-ci.

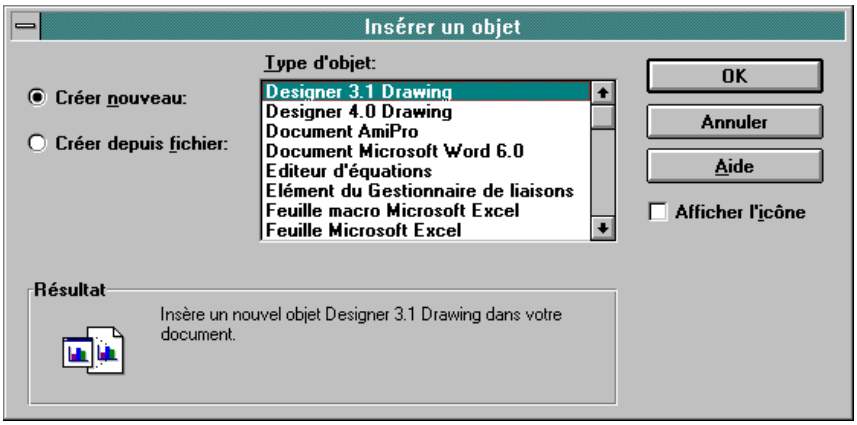
Pour ce faire on pourra utiliser une des propriétés ObjClass, ObjDoc,ObjItem (cette dernière donnant le meilleur résultat).

Selon que l'on souhaite créer un objet lié ou un objet imbriqué la procédure d'initialisation est différente.

Cas d'un objet imbriqué :

Dans ce cas il est préférable que l'objet à imbriqué n'ait pas été créé au préalable. Sa création sera donc réalise selon le mécanisme suivant :

1. Sélectionner la propriété ObjClass du composant TOleContainer. Cliquer sur le bouton (...) de manière à faire apparaître la boite de dialogue 'Insérer un objet'.



2. Sélectionner l'option 'Créer nouveau' et le type d'objet OLE à créer.

3. A la fermeture de la boite de dialogue, le composant est prêt. En double-cliquant dessus on lance le serveur OLE adéquat dans lequel il va falloir créer l'objet OLE.

4. Fermer l'application serveur. Revenir dans l'application en cours de conception. Eventuellement redimensionner le composant TOleContainer.

5. Exécuter.

Par exemple :

En sélectionnant un objet de type 'Feuille Microsoft Excel 5.0' on appelle Excel 5.0 dans lequel on va pouvoir créer une feuille de calcul (légendes, affichages divers et formules de calcul).

En fermant Excel, la feuille de calcul créée apparaît dans le composant TOleContainer. Redimensionner le composant si nécessaire pour afficher toutes les données nécessaires.

A l'exécution la feuille de calcul apparaît dans le composant. Le fait de double-cliquer sur celui-ci lance l'application serveur dans laquelle on peut réaliser les modifications de données souhaitées.

Dans ce mode de fonctionnement, il n'y a pas de fichier contenant les données sur le disque. Elles sont toutes insérées dans le code de l'application (on s'en rend compte lorsqu'on sort de l'application serveur : même si on a modifié les données, l'application se ferme sans demander de confirmation d'enregistrement des modifications).

Dans ce mode de fonctionnement seule l'application cliente -via un composant TOleContainer - peut accéder aux données. On ne peut donc pas y accéder par ailleurs (par un lien DDE par exemple). En conséquence on ne peut utiliser d'objet Ole imbriqué que lorsque l'on n'a pas besoin de récupérer la (ou les) données modifiées pour la suite de l'application.

Les modifications apportées aux données (via l'activation de l'application serveur), ne sont pas sauvegardées d'une exécution à l'autre de l'application. Pour assurer cette sauvegarde il faut utiliser les méthodes SaveToFile() et LoadFromFile() du composant TOleContainer.

Exemple :

Dans le gestionnaire d'événement OnCreate de la feuille insérer le code :
SaveToFile ('c:\delphi\essais\dde\balance.xls') ;

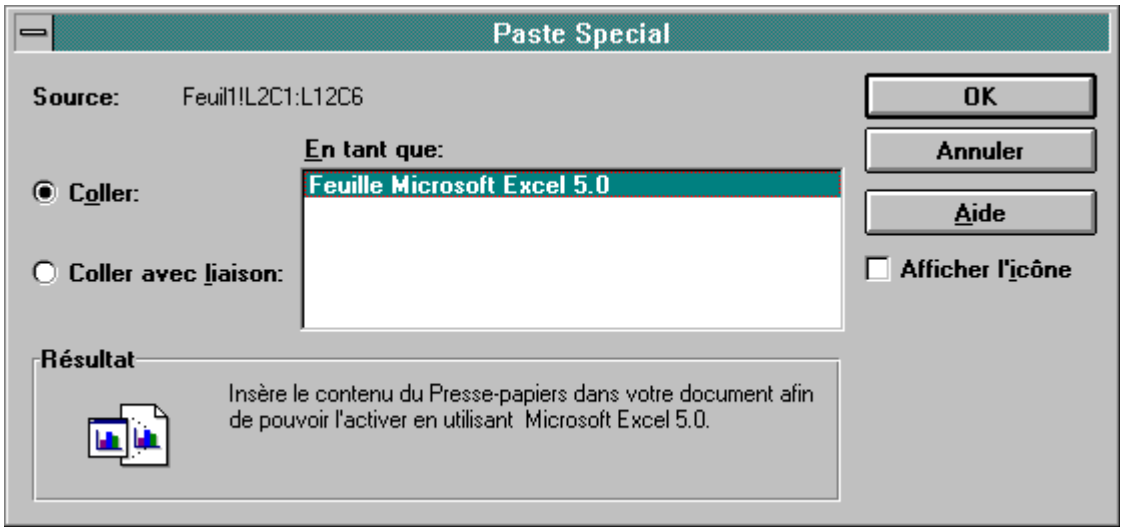
Dans le gestionnaire d'événement associé à OnClose insérer le code :
LoadFromFile ('c:\delphi\essais\dde\balance.xls') ;

- Même si le fichier de sauvegarde utilise une extension conforme à celles utilisées par l'application serveur, cette dernière n'est pas en mesure de lire directement son contenu : en cas d'essai de lecture directe par le serveur un message d'erreur est affiché.

Cas d'un objet lié :
Dans ce cas il est préférable que l'objet à lier ait été créé au préalable dans l'application serveur.

La création du lien OLE se fait alors selon la procédure suivante :

1. Créer le document à lier. Sélectionner la partie du document que l'on souhaite lier. La copier dans le presse-papiers.
2. Revenir dans l'application container sous DELPHI.
3. Sélectionner la propriété ObjItem du composant TOleContainer. Double-cliquer sur le bouton (...) . Une boite de dialogue 'Collage spécial' apparaît.



4. Choisir l'option 'Coller avec liaison' et valider.
5. Lorsque la boite de dialogue se ferme, l'objet container est lié avec le fichier souhaité. Toutes les modifications dans le fichier (par l'application ou par une autre application sera répercutée dans l'application. Les sauvegardes sont réalisées par l'application serveur.

Exemple :
Créer dans Excel 5, une feuille de calcul nommée BALANCE.XLS.
Copier l'ensemble des cellules réellement utilisées. Dans DELPHI, initialiser la propriété ObjItem en lançant la boite de dialogue adéquate. Coller avec liaison.

A l'exécution le fait de double-cliquer sur la feuille de calcul lance Excel 5 dans lequel on peut effectuer les modifications voulues. En sortant d'Excel (avec demande d'enregistrement) on revient dans l'application où les valeurs ont été modifiées.

> Un lien OLE utilisant le 'collage avec lien', pour assurer la mise à jour automatique des données au fur et à mesure des modifications, est très gourmand en ressource. Il ne faut donc pas abuser de ce type de liens.

RÉCUPÉRATION D'UNE DONNÉE MODIFIÉE

Le problème majeur d'un lien OLE vient du fait que, s'il permet de modifier plusieurs données et réaliser des opérations complexes sur des données (via l'activation de l'application serveur), il ne permet pas de récupérer celles-ci dans l'application cliente afin que les modifications soient prises en compte dans la suite de l'exécution.

Une solution consiste à mettre en oeuvre un lien DDE conjointement avec le lien OLE. Cette solution, séduisante en soit, ne peut être réalisée qu'en prenant certaines précautions:

Il faut que l'objet OLE soit un objet lié (il faut qu'il y ait un fichier externe pour créer le lien DDE).

Il faut coordonner finement l'ouverture du lien DDE avec celle du lien OLE :

- Il faut initialiser la propriété ConnectMode du composant TDDEClientConv à ddeManual de manière à ce que le lien DDE ne soit pas créé dès la création de la feuille.

- Il ne faut créer le lien DDE que si le lien OLE a été créé (par défaut il est créé par double-click sur le composant TOleContainer).

- Dans ce cas il faut, encore une fois, être sûr que l'application serveur soit complètement chargée en mémoire avant d'ouvrir le lien. Sinon la tentative d'ouverture par la méthode Setlink () du composant TDDEClientConv se soldera par un échec. La solution utilisée (mais non fiable 100 %) est celle qui consiste à mettre en oeuvre un Timer qui ne se déclenche que lorsque le lien OLE est mis en oeuvre.

Exemple :
En reprenant le fichier BALANCE.XLS précédent dont les données sont liées par un lien OLE 'lié' on rajoute les composants nécessaires à la création d'un lien DDE (TDDEClientConv et TDDEClientItem) , un composant TEdit et un composant TTimer que l'on initialise comme suit :

DdeClientConv1 :	***Connectmode := ddeManual***
DdeClientItem1 :	***DdeConv := DdeClientConv1***
Timer1 :	***Enabled := False***
	Interval := 5000
Edit1 :	***Text := ''***
	Visible := False

Les gestionnaires d'événements créés sont alors :

```
procedure TForm1.OleContainer1DblClick(Sender: TObject);
begin
        Timer1.Enabled := True ;
end;
```

Cet événement survient lorsque l'on active le lien OLE. Le serveur EXCEL est alors chargé en mémoire et les données sont affichées dans le container. Pendant ce temps, le Timer est déclenché. Au bout de 5 secondes il va exécuter le gestionnaire d'événement qui lui est associé

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
        with DDEClientConv1 do
        begin
                if SetLink('Excel','C:\DELPHI\ESSAIS\DDE\
                [BALANCE.XLS]Feuil1')then
                begin
                        Screen.Cursor := crHourGlass ;
                        DdeClientItem1.DdeItem := 'L12C6' ;
                        Openlink ;
                        Edit1.Visible := True ;
                end
                else
                        MessageDlg('Lien non établi', mtInformation, [mbOK], 0);
        end ;
        Timer1.Enabled := False ;
        { Désactivation du Timer de manière à ce qu'il ne relance pas le lien DDE }
end;

procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
        Edit1.Text := DdeClientItem1.Text ;
        Screen.Cursor := crDefault ;
end;
```

Si le lien DDE a pu être établi, les modifications apportées dans le serveur OLE sont répercutées directement dans la cellule liée par le lien DDE. La dernière information fournie est conservée dans le composant TEdit lorsque le lien OLE est fermé

OLE AUTOMATION

L'automatisation OLE est un protocole par lequel une application peut accéder à un objet résidant dans une autre application ou une DLL. Ce protocole vous permettra de :

- Contrôler les actions d'une application ou d'une DLL.
- Accéder aux fonctionnalités d'une application ou d'une DLL.

Une application qui peut être automatisée est appelée un **serveur d'automatisation**. Une application qui automatise une autre application est appelée un **contrôleur d'automatisation** ou client.

Cette section montre comment utiliser des objets Delphi de haut niveau pour créer des clients d'automatisation OLE. Créer des clients d'automatisation OLE n'est pas difficile. Cependant,

cette technologie a de nombreuses implications qui rendent le sujet plus complexe qu'il n'y paraît de prime abord. C'est pourquoi, cette section étudie un certain nombre de principes directeurs de l'architecture OLE nécessaires au fonctionnement à grande échelle de cette technologie.

Les sujets suivants seront exposés :

- Création des objets OLE à l'aide de la fonction CreateOleObject
- Utilisation des variants pour encapsuler des objets OLE
- Clients OLE
- Tableaux de variants

L'automatisation OLE vous permet d'accéder à des objets qui résident non seulement dans votre programme, mais également dans les autres programmes se trouvant sur votre système. Plus précisément, vous pouvez accéder aux méthodes et aux propriétés de ces objets, mais non à leurs données brutes. Et vous pouvez le faire sans tenir compte du langage de programmation qui a été utilisé pour implémenter l'objet. Avec l'arrivée de Network OLE, les programmeurs ont pu étendre ce principe de fonctionnement à un ensemble de machines connectées en réseau et créer ainsi des applications partitionnées.

Il y a deux types majeurs d'automatisation OLE :

- Les serveurs d'automatisation OLE
- Les contrôleurs d'automatisation OLE (ou clients)

Les serveurs d'automatisation fournissent les fonctionnalités auxquelles accèdent les contrôleurs d'automatisation. En d'autres termes, l'application ou la DLL qui est l'hôte d'un objet est appelée le serveur, et l'application ou la DLL qui y accède est appelée le contrôleur. Delphi vous permet d'intégrer et de faire communiquer vos applications et vos DLL avec d'autres applications, en tant que serveurs d'automatisation ou en tant que contrôleurs OLE.

Parmi les exemples classiques de serveurs d'automatisation OLE, citons Word et Excel de Microsoft. Ces deux applications peuvent être pilotées par une application Delphi et par tout autre contrôleur d'automatisation. Pour plus d'informations sur les serveurs d'automatisation OLE, reportez-vous à "Création de serveurs OLE Automation" dans le Guide du développeur.

L'exemple classique d'une automatisation OLE est celui du pilotage de Word de Microsoft. Si vous avez la version Word 6.X ou Word 7.X sur votre système, vous pourrez y accéder depuis une application Delphi en entrant le code suivant :

```
uses OleAuto;
procedure TForm1.Button1Click(Sender: TObject);
var
  V: Variant
begin
 V := CreateOleObject('Word.Basic');
 V. Insertion('Le bonjour de la part de Delphi');
end;
```

Ce code insère les mots 'Le bonjour de la part de Delphi' dans un document Word existant. Word doit avoir été démarré et un document doit être ouvert pour que ce code puisse fonctionner. Vous verrez ultérieurement comment prendre en compte la situation où Word n'est pas encore démarré.

Il y a trois éléments clés dans le code ci-dessus. Le premier est l'unité **OleAuto** qui apparaît dans la clause uses. Cette unité contient l'essentiel du code permettant de gérer l'automatisation à partir d'une application Delphi.

Les variants ont été introduits dans Pascal Objet car Microsoft en fait une utilisation intensive dans le code ayant un rapport avec l'automatisation OLE. Ils ne sont pas d'une nécessité absolue, mais ils simplifient considérablement l'automatisation OLE, compte tenu de l'implémentation OLE définie par Microsoft :

V := CreateOleObject('Word.Basic');

Le code ci-dessus affecte un objet OLE au variant V. L'objet OLE en question réside à l'intérieur de Word. Il s'agit plus précisément d'un serveur d'automatisation OLE qui réside à l'intérieur de Word.

Le troisième élément clé est l'appel à la méthode Insertion de Word Basic :

V.Insertion('Bonjour de la part de Delphi');

Insertion n'est pas une méthode ou une fonction Pascal Objet, ni une partie de l'API Windows. Elle appartient à Word, et vous êtes en mesure de l'appeler directement depuis une application Delphi grâce à l'automatisation OLE.

Lorsque vous appelez CreateOleObject, vous lui transmettez une chaîne. Celle-ci contient une entité appelée un ID de programme ou ProgID. Word.Basic est le ProgID du serveur d'automatisation OLE de Microsoft Word. L'identifiant d'un programme est une chaîne qui peut être recherchée dans la base de registres, et qui fait référence à un CLSID. Les CLSID sont des nombres, statistiquement uniques, utilisés par le système d'exploitation pour faire référence à un objet OLE.

Lorsque vous appelez CreateOleObject, Windows vérifie que Word est déjà chargé en mémoire, et, si cela n'est pas le cas, il démarre le programme en lui transmettant la ligne de commande ci-dessus. Il recherche ensuite l'interface (en fait un pointeur) de l'objet d'automatisation OLE demandé et renvoie celle-ci comme résultat de l'appel à CreateOleObject. Encore une fois, ce n'est pas un pointeur sur Word lui-même que vous récupérez, mais un pointeur sur un objet qui réside dans Word.

Une fois récupérée l'interface de l'objet automatisation OLE de Word, vous pouvez appeler les fonctions rendues accessibles par cet objet. Dans notre exemple, toutes les fonctions Word Basic décrites dans l'aide en ligne de Word deviennent accessibles. Vous accédez ainsi à quelque 200 fonctions incluant notamment des commandes pour ouvrir, sauvegarder, formater ou imprimer des documents.

Pour insérer du texte dans un document alors que Word n'est pas exécuté, faites appel au code suivant :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 V := CreateOleObject('Word.Basic');
 V.FichierNouveau('Normal');
 V.Insertion('L'utilisation d'OLE Automation' + #13);
 V.Insertion('apporte de nombreux avantages à vos applications.');
 V.FichierEnregistrerSous('C:\MonFichier.DOC');
end;
```

Le point intéressant concernant le code ci-dessus est le suivant : Word n'apparaît jamais sur l'écran. Word est démarré temporairement en mémoire et reste invisible. Dès que la variable V devient hors portée ou dès qu'elle devient varNothing, l'exécution de Word s'arrête. Mis à part le bruit de votre disque dur, le seul moyen d'être sûr du fonctionnement du code ci-dessus consiste à démarrer Word après l'exécution de votre application et de tenter d'ouvrir MonFichier.DOC pour vérifier son contenu.

Le code ci-dessus commence par un appel à CreateOleObject. Cette ligne a pour effet de charger Word en mémoire si cette application n'est pas déjà démarrée. Il la démarre, mais en lui transmettant le paramètre /Automation. Le programme Word sait que lorsqu'il reçoit ce paramètre au démarrage, il doit s'exécuter silencieusement en tâche de fond sans jamais devenir visible à l'écran. Il arrête son exécution dès qu'il n'a plus aucun client d'automatisation OLE à servir.

Notez que vous pouvez ajouter un retour chariot dans ces lignes en insérant un caractère retour à la ligne (#13) dans le texte.

Ce n'est pas par hasard que Word s'exécute en tâche de fond quand il est appelé en mémoire par un client d'automatisation OLE. En fait, cela fait partie des directives qui définissent un serveur : il doit être capable de s'exécuter silencieusement en tâche de fond sauf s'il est directement appelé par un utilisateur plutôt que par un contrôleur d'automatisation OLE.

Delphi autorise la création de tableaux de variants qui sont la version Delphi des "tableaux sûrs" utilisés dans l'automatisation OLE. Les tableaux de variants sont l'implémentation Delphi des tableaux sûrs.

Les tableaux de variants (tableaux sûrs) sont coûteux en terme de mémoire et de cycles CPU, c'est pourquoi ils ne sont pas utilisés sauf dans du code d'automatisation OLE, et dans certaines situations spécifiques où ils offrent un avantage incontestable par rapport aux tableaux standard. Par exemple, certaines parties du code de la gestion des bases de données font appel aux tableaux de variants.

La littérature OLE se réfère à des tableaux sûrs car ils contiennent des informations concernant le nombre de leurs dimensions, et les bornes de chacune d'elles. Le fichier

Windows OLEAUTO32.DLL contient toute une série d'appels de type SafeArrayX permettant de manipuler ces tableaux.

Delphi encapsule les appels SafeArrayX dans plusieurs fonctions. Les plus importantes sont VarArrayCreate et VarArrayOf. Elles sont utilisées pour créer des tableaux de variants.

La déclaration de VarArrayCreate ressemble à ceci :

function VarArrayCreate(const Bounds: array of Integer; VarType: Integer): Variant;

Le paramètre Bounds définit les dimensions du tableau. Le paramètre VarType définit le type de la variable stockée dans le tableau.

Un tableau de variants à une dimension peut être alloué de la façon suivante :

MonVariant := VarArrayCreate([0, 5], varVariant);

Ce tableau a six éléments, chaque élément étant lui-même un variant. Vous pouvez affecter un tableau de variants à un ou plusieurs éléments de ce tableau. De cette façon, vous pouvez avoir des tableaux à l'intérieur de tableaux, eux-mêmes à l'intérieur de tableaux, si vous le souhaitez.

Si vous connaissez le type des éléments à utiliser dans un tableau, vous pouvez définir ainsi le paramètre VarType. Par exemple, si vous savez que vous allez travailler sur des entiers, vous pouvez écrire :

MonVariant := VarArrayCreate([0, 5], varInteger);

Vous ne devez pas utiliser varString comme deuxième paramètre, mais plutôt varOleStr. N'oubliez pas qu'un tableau de variants peut occuper jusqu'à 16 octets en mémoire pour chacun de ses membres, alors que les autres types occupent un espace plus restreint.

Les tableaux de variants peuvent être redimensionnés à l'aide de la fonction VarArrayRedim :

procedure VarArrayRedim(var A: Variant; HighBound: Integer);

La variable à redimensionner est transmise comme premier paramètre et le nombre d'éléments que le tableau doit contenir est transmis comme deuxième.

Un tableau à deux dimensions est déclaré comme ceci :

MonVariant := VarArrayCreate([0, 5, 0, 5], varVariant);

Ce tableau a deux dimensions, chacune contenant 6 éléments. Pour accéder à un membre de ce tableau, écrivez ceci :

```
procedure TForm1.GridClick(Sender: TObject);
var
 MonVariant: Variant;
begin
 MonVariant := VarArrayCreate([0, 5, 0, 5], varVariant);
 MonVariant[0, 1] := 42;
 Form1.Caption := MonVariant[0, 1];
end;
```

Notez que le tableau effectue les conversions de types nécessaires car il s'agit d'un tableau de variants et non d'entiers.

Vous pouvez utiliser la routine VarArrayOf pour construire rapidement un tableau de variants à une dimension :

```
function VarArrayOf(const Values: array of Variant): Variant;
```

La fonction appelle de façon interne VarArrayCreate en lui transmettant un tableau Variant comme premier paramètre et varVariant comme deuxième paramètre. Voici un appel typique à VarArrayOf :

```
V := VarArrayOf([1, 2, 3, 'Total', 5]);
```

L'extrait de code suivant montre comment utiliser la fonction VarArrayOf.

```
procedure TForm1.ShowInfo(V: Variant);
begin
 Caption := V[3];
end;
procedure TForm1.Button1Click(Sender: TObject);
var
 V: Variant;
begin
 V := VarArrayOf([1, 2, 3, 'Quatre', 5]);
 ShowInfo(V);
end;
```

Ce code affiche le mot "Quatre" sur la barre de titre de Form1.

La fonction ShowInfo montre comment travailler avec un tableau de variants transmis par une fonction OLE ou toute autre routine. Notez qu'il n'y a rien à faire de particulier pour accéder à un variant en tant que tableau. Si vous tentez de transmettre à cette fonction un variant avec une valeur VType de varInteger, cela provoque une exception car vous cherchez à traiter le variant comme un tableau. En résumé, le variant doit avoir un VType qui vaut VarArray pour que l'appel à ShowInfo réussisse. Vous pouvez utiliser la fonction VarType pour vérifier le paramétrage en cours du VType d'un variant ou bien appeler VarIsArray, qui renvoie une valeur booléenne.

Vous pouvez utiliser les fonctions VarArrayHighBound, VarArrayLowBound et VarArrayDimCount pour déterminer le nombre de dimensions de votre tableau et les bornes de chacune d'elles. La fonction suivante affiche une boîte de message indiquant le nombre de dimensions d'un tableau de variants, ainsi que la valeur des bornes supérieures et inférieures pour chacune de ses dimensions :

```
procedure TForm1.ShowInfo(V: Variant);
var
 Count, HighBound, LowBound, i: Integer;
 S: string;
begin
 Count := VarArrayDimCount(V);
 S := #13 + 'Nombre de dimensions : ' + IntToStr(Count) + #13;
 for i := 1 to Count do
 begin
   HighBound := VarArrayHighBound(V, i);
   LowBound := VarArrayLowBound(V, i);
   S := S + 'Borne supérieure : ' + IntToStr(HighBound) + #13;
   S := S + 'Borne inférieure : ' + IntToStr(LowBound) + #13;

 end;
 ShowMessage(S);
end;
```

Cette routine commence par récupérer le nombre de dimensions du tableau. Elle les parcourt ensuite l'une après l'autre, en récupérant les bornes supérieures et inférieures. Si vous aviez créé un tableau avec l'appel suivant :

```
MonVariant := VarArrayCreate([0, 5, 1, 3], varVariant);
```

la fonction ShowInfo produirait la sortie ci-dessous, dans le cas où MonVariant lui serait transmis :

```
Nombre de dimensions : 2
Borne supérieure : 5
Borne inférieure : 0
Borne supérieure : 3
Borne inférieure : 1
```

ShowInfo déclencherait une exception si vous lui transmettiez un variant qui obligerait VarIsArray à renvoyer False.

Les tableaux de variants entraînent un surcroît de travail. Pour simplifier le traitement des tableaux, vous pouvez utiliser les deux fonctions VarArrayLock et VarArrayUnlock. La première renvoie un pointeur sur les données stockées dans le tableau. VarArrayLock accepte un tableau de variants en entrée et renvoie un tableau Pascal Objet standard. Pour que cela fonctionne, le tableau doit être explicitement déclaré avec un type standard tel que Integer, Bool, string, Byte ou Float. Le type utilisé dans le tableau de variants et celui utilisé dans le tableau Pascal Objet doivent être identiques pour tous leurs membres.

Voici un exemple d'utilisation de VarArrayLock et VarArrayUnlock :

```
const
 HighVal = 12;
function GetArray: Variant;
var
 V: Variant;
 i, j: Integer;
begin
 V := VarArrayCreate([0, HighVal, 0, HighVal], varInteger);
 for i := 0 to HighVal do
   for j := 0 to HighVal do
     V[j, i] := i * j;
 Result := V;
end;
procedure TForm1.LockedArray1Click(Sender: TObject);
type
 TData = array[0..HighVal, 0..HighVal] of Integer;
var
 i, j: Integer;

 V: Variant;
 Data: ^TData;
begin
 V := GetArray;
 Data := VarArrayLock(V);
 for i := 0 to HighVal do
   for j := 0 to HighVal do
     Grid.Cells[i, j] := IntToStr(Data^[i, j]);
 VarArrayUnLock(V);
end;
```

Notez que ce code commence par verrouiller le tableau, puis y accède comme s'il s'agissait d'un pointeur sur un tableau standard. Pour finir, il restitue le tableau quand l'opération se termine. Vous ne devez pas oublier d'appeler VarArrayUnlock quand vous avez fini de travailler avec les données du tableau :

```
Data := VarArrayLock(V);
for i := 0 to HighVal do
  for j := 0 to HighVal do
```

L'une des meilleures raisons d'utiliser un tableau de variants est le transfert de données binaires entre votre application et un serveur. Prenons l'exemple d'un fichier binaire tel qu'un fichier WAV ou AVI : vous pouvez le faire passer de votre application vers un serveur OLE et réciproquement à l'aide de tableaux de variants. Dans une telle situation, il est parfaitement légitime d'utiliser VarArrayLock et VarArrayUnlock. Bien évidemment, vous devez utiliser VarByte comme deuxième paramètre de VarArrayCreate lors de la création du tableau. Autrement dit, vous travaillez sur un tableau d'octets en y accédant directement après l'avoir

verrouillé et en plaçant et en retirant des données dans la structure. De tels tableaux ne sont sujets à aucune conversion lorsqu'ils sont véhiculés au-delà des frontières d'applications.

Souvenez-vous que les tableaux de variants ne doivent être utilisés que dans des circonstances précises. Il s'agit d'outils particulièrement utiles pour les appels à des objets d'automatisation OLE. Toutefois, ils sont plus lents à traiter et plus encombrants que les tableaux standard Pascal Objet, et ne doivent donc être utilisés que s'ils sont nécessaires.

DONNÉES ACCESSIBLES PAR LE RÉSEAU

Les développements d'applications professionnelles se font de plus en plus en prenant en compte la dimension "réseau". Cela peut se faire par simple partage de fichiers (accessibles directement dès lors qu'un gestionnaire de réseau a été installé sur les différentes machines du réseau). Cela met souvent en oeuvre un mécanisme beaucoup plus élaboré, dit mécanisme "client/serveur ", dans lequel les traitements sont partagés entre une machine serveur - généralement chargée de gérer les accès aux données en assurant leur sécurité et leur intégrité - et une ou plusieurs machines clientes chargées plus spécialement d'assurer le dialogue interactif avec les utilisateurs et la présentation des données.

Serveur de fichiers et serveur de traitement :

Dans tout ce qui suit, il y aura lieu de distinguer la notion de serveur de fichiers et celle de serveur de traitement.

Dans le premier cas la notion de serveur se réduit à la capacité à partager des fichiers stockés d'une manière centralisée et passive sur un disque distant. Tout le traitement est réalisé au niveau de l'application qui demande l'accès par transfert du fichier du serveur vers l'application pour traitement.

Dans le deuxième cas, celui qui correspond au modèle "client / serveur", une partie plus ou moins importante du traitement souhaité est réalisée par une application s'exécutant sur la machine serveur. L'application cliente communique ses demandes par des requêtes. L'application serveur renvoyant des résultats.

Dans ce chapitre nous allons donc étudier les différentes manières d'accéder aux données stockées "quelque part" sur le réseau. Il est impératif de noter que les mécanismes mis en oeuvre ne sont pas strictement circonscrits à l'aspect "programmation" (et donc à DELPHI) : dans la plupart des cas il faudra au préalable passer par une phase de configuration du système et des différents intervenants dans le lien à mettre en place qui peut se révéler relativement longue et ardue.

ACCÈS AUX DONNÉES PAR PARTAGE DE FICHIER

Il est possible d'accéder facilement aux données stockées, le cas échéant dans différentes tables, sur une machine distante en utilisant les possibilités de partage fournies par un gestionnaire de réseau simple (par exemple : Windows for Workgroups).
Pour cela il suffit de configurer le partage du répertoire dans lesquels les différentes tables sont stockées en utilisant le gestionnaire de fichiers de Windows puis faire reconnaître le chemin ainsi constitué par l'utilitaire de configuration de BDE afin de créer un alias.

CONFIGURATION DU SYSTÈME

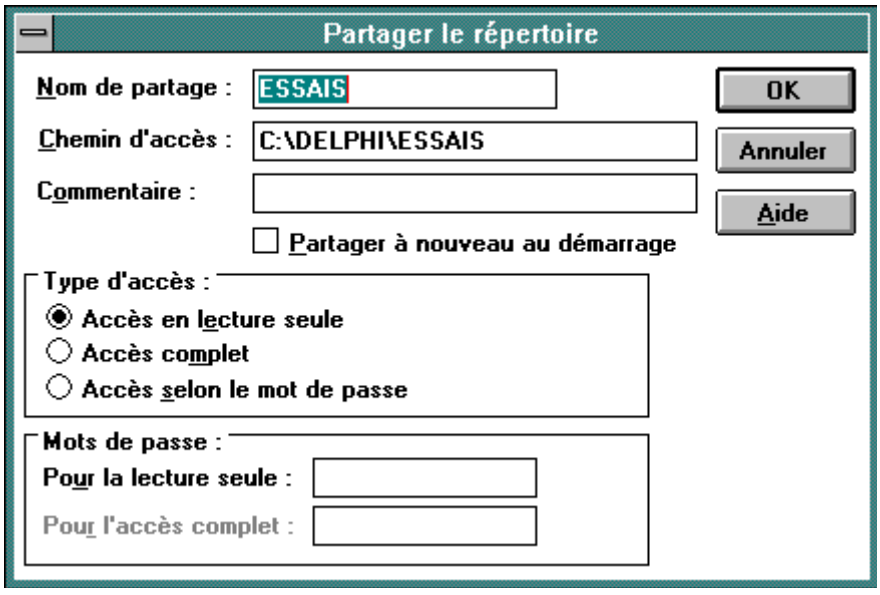
Pour connecter un lecteur distant au système de fichiers de la machine cliente il faut :

Sur la machine où sont stockés les fichiers :

- Sélectionner le répertoire "à partager" dans le gestionnaire de fichiers.
- Partager ce répertoire en utilisant le menu ' Disque | Partager...'. Une boite de dialogue apparaît permettant de configurer ce partage.

La boite de dialogue permet, entre autres, un partage en lecture seule ou en accès complet (option à utiliser si l'on souhaite pouvoir modifier les données) avec éventuellement utilisation d'un mot de passe par l'utilisateur souhaitant accéder au répertoire.

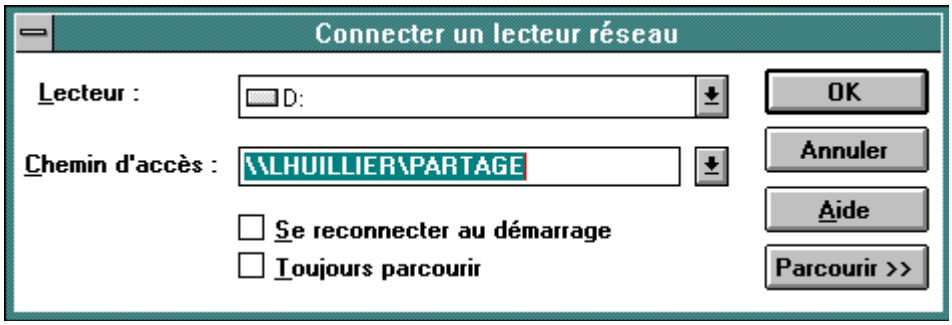
Une option intéressante permet d'automatiser le partage à chaque démarrage de la machine.



Boite de dialogue permettant le partage d'un répertoire.

Sur la machine où se trouve l'application "cliente"

- Utiliser le menu 'Disque | Connecter un lecteur réseau... ' pour assurer le lien avec la machine distante. Une boite de dialogue apparaît. Elle propose la lettre sous laquelle le lecteur distant sera répertorié dans le gestionnaire de fichiers. Il faut ensuite désigner (éventuellement en parcourant la liste des répertoires partagés du réseau) le répertoire auquel on souhaite accéder.



L'option 'Se reconnecter au démarrage' est intéressante car elle permet d'automatiser la procédure à chaque démarrage.

A partir de ce moment, le répertoire partagé est accessible au même titre que les autres répertoires du système de fichiers local.

CONFIGURATION DE L'APPLICATION CLIENTE

Une fois le lecteur distant connecté, il est possible d'y faire référence à partir de l'application DELPHI cliente.

Plusieurs méthodes sont disponibles :

Mise en oeuvre d'un alias :

Dans l'utilitaire 'Configuration Database Engine ' il est possible de créer un alias indiquant, dans sa propriété PATH, le chemin d'accès au répertoire partagé.

A partir de là on peut concevoir une application DELPHI utilisant cet alias, dans les mêmes conditions que lorsqu'il s'agissait de concevoir une application locale.

- Si l'accès semble simple à réaliser, il n'est cependant pas garanti. En effet il se peut que la connexion au lecteur partagé ne soit pas réalisée ou alors qu'elle le soit avec une autre lettre de référence pour le lecteur partagé (E: au lieu de D: par exemple). Dans ces conditions l'alias est inutilisable et l'accès à distance impossible.

Initialisation de la propriété DatabaseName :

Il est possible d'initialiser la propriété DatabaseName des composants TTable ou TDatabaseName en fournissant, à l'exécution, le chemin réel indiquant le répertoire partagé. Là aussi plusieurs méthodes peuvent être mises en oeuvre :

On peut se contenter de proposer à l'utilisateur une zone de saisie dans laquelle il devra entrer le chemin complet d'accès (ce qui présuppose une connaissance correcte de Windows).

La connexion n'est alors réalisée que sur ordre de l'utilisateur (click sur un bouton après saisie du chemin dans une zone de saisie par exemple). On peut alors avoir le code suivant :

```
procedure TForm1.Button1Click (Sender: TObject);
begin
        Table1.DatabaseName := Edit1.text ;
                        { Chemin du répertoire partagé }
        Table1.Tablename := 'BIBLIOTH.DB' ;
                        { Nom de la table cible }
        Table1.Open ;
end;
```

Cette méthode permet d'assurer la connexion si le chemin indiqué dans la zone de saisie est correct... et que le lecteur est bien partagé.

Une version améliorée de ce code est la suivante :

```
procedure TForm1.Button1Click (Sender: TObject);
var
        Fichier : string [ 100 ] ;
begin
        Fichier := Edit1.Text + '\D0123456.TXT' ;
        if FileExists( Fichier ) then
        begin
                Table1.DatabaseName := Edit1.text ;
                Table1.TableName := 'BIBLIOTH.DB' ;
                Table1.Open ;
        end
        else
                MessageDlg ( 'Le chemin spécifié est incorrect ',mtError,[mbOK],
0 );
end ;
```

Le fichier DO1234566.TXT est un fichier "drapeau" créé spécifiquement pour être installé dans le répertoire partagé. Le test de l'existence de ce fichier permet de s'assurer qu'on accède bien au bon répertoire partagé

Automatisation de la procédure de connexion

Pour que l'application soit pleinement satisfaisante il faut, d'une part, renforcer les tests et, d'autre part, automatiser le plus possible la procédure d'accès à distance. En utilisant certaines fonctions de gestion du système de fichiers proposées par DELPHI on arrive à une solution satisfaisante.

Le code résultant, exécuté au démarrage de l'application sans intervention de l'utilisateur, est alors le suivant :

```
procedure TForm1.FormCreate(Sender: TObject);
var
        NumLecteur : integer ;
        { Numéro du lecteur : A = 0 , B = 1 , C = 2, D = 3 , etc .... )
        LettreLecteur : char ;                  { Lettre de dénomination }
        Test : string [ 100 ] ;
        { Chemin d'accès au fichier test D0123456.DPH }
        trouve : boolean ;              { vrai si le fichier test a été trouvé }
        partage : boolean ;             { vrai si un lecteur est partagé }
begin
        NumLecteur := 2 ;               { On débute par le test du disque C: }
        trouve := false ;
        partage := false ;
        while ( not trouve ) AND ( NumLecteur <= 25 ) do
        { tant qu'on n'a pas trouvé le bon lecteur, ou qu'on est pas arrivé à Z }
        begin
                if ( GetDriveType ( NumLecteur ) = DRIVE_REMOTE ) then
                { Si le disque est de type "éloigné " }
                begin
                        partage := true ;
                        LettreLecteur := Chr ( NumLecteur + 65 ) ;
                        { conversion N° -> lettre }
                        Test := LettreLecteur + ':\tables\D0123456.dph' ;
                        { création chemin }
                        if FileExists( Test ) then      { test d'existence }
                        begin   { connexion }
                        Table1.DatabaseName := ExtractFilePath ( Test ) ;
                                Table1.TableName := 'BIBLIOTH.DB' ;
                                Table1.Open ;
                                trouve := True ;
                        end ;
                end ;
                inc ( NumLecteur ) ; { Test du lecteur suivant }
        end ;
        if ( not trouve ) and ( not partage ) then
                MessageDlg ( 'Il n''y a pas de lecteur partagé' + #10 +
                        ' Configurez votre système', mtError , [mbOK] , 0 ) ;
end;
```

MÉCANISMES DE VERROUILLAGE

A partir du moment où l'on réalise une application utilisant des données stockées sur une autre machine, il peut se poser la question de l'utilisation simultanée de la même table, voire du même enregistrement, par plusieurs utilisateurs.

En effet rien n'interdit que au moment où l'on accède à une donnée, un autre utilisateur réalise la même opération sur la même donnée et la modifie à sa convenance. Dans ces conditions quelle est la vrai valeur de la donnée affichée, et quelle est celle de la donnée modifiée ?

Pour éviter de lire des données qui ne sont plus valides au moment de la lecture ou de modifier en même temps la même donnée, il faut donc mettre en place des mécanismes qui permettent de s'assurer qu'un utilisateur est le seul à accéder, à un moment précis, à une donnée.

Ces mécanismes algorithmiques vont du simple verrouillage de la table ou de l'enregistrement à la mise en oeuvre de véritables procédures transactionnelles qui permettent de s'assurer que tous les ordres (en particulier ceux qui génèrent une modification de données) envoyés sont bien pris en compte, et surtout que si la transaction est interrompue les données sont remises dans leur état initial.

Action par défaut

A partir du moment où un utilisateur se connecte à une table stockée sur une machine distante (selon les procédés vus précédemment) le BDE crée deux fichiers dans le répertoire partagé (PARADOX.LCK et PDOXUSRS.LCK).

Lorsqu'un autre utilisateur cherche à se connecter à la même table, un message d'erreur Windows, l'informant de l'utilisation de la table, est affiché. La connexion n'est pas établie.

En fait l'impossibilité de pouvoir accéder en même temps à la même table vient du fait que Paradox met en place un fichier PDOXUSRS.NET lorsque l'on accède à une table via le réseau. Par défaut ce fichier est créé dans le répertoire spécifié dans le module 'Configuration Database Engine', onglet Paradox, valeur NetDir.

Il est évident que chaque machine possède un répertoire NetDir qui lui est local. De ce fait il existe plusieurs fichiers PDOXUSRS.NET sur le réseau d'où le libellé du message d'erreur :

' Plusieurs fichiers .NET sont utilisés. Fichier < nom du fichier .LCK concerné > .

Pour que l'on puisse accéder simultanément à une même table il faut que le fichier .NET soit unique et lisible par toutes les applications clientes. La solution évidente est donc de le placer dans le répertoire partagé où se situe les tables à utiliser. (ou pour le moins dans la racine du disque partagé).

On peut réaliser ceci en utilisant l'utilitaire 'Configuration DataBase Engine'. Mais cela veut dire qu'il faut configurer toutes les machines clientes et que, par dessus tout, le chemin indiqué risque de ne pas être le bon, selon le nombre de lecteurs connectés au moment de l'exécution de l'application.

Il faut donc réaliser l'initialisation de la propriété NetDir de Paradox à l'exécution en lui fournissant le nom du répertoire partagé. Pour cela il faut utiliser le composant TSession.

Dans le cas qui nous intéresse, c'est la propriété NetDirFile de Session qui doit être initialisée.

Le composant TSession

Le composant TSession est un composant invisible, dont une instance nommée Session est créée automatiquement par DELPHI dès lors qu'on réalise une application accédant à une base de données (il est du même niveau que les composants TApplication,TScreen, etc ...).

Session permet de contrôler d'une manière globale les connexions à la base de données. Il contient un certain nombre de propriétés et de méthodes qui permettent de réaliser le contrôle fin d'une base de données.

Il suffit donc de rajouter la ligne d'instruction :

```
.......
if FileExists( Test ) then
begin
Session.NetFileDir := ExtractFilePath (Test ) ;
        Table1.DatabaseName := ExtractFilePath ( Test ) ;
        ......
end ;
```

▪ L'initialisation de Session doit être réalisée avant toute ouverture de table.

A partir de ce moment, la première application accédant à la table partagée créera un fichier .NET dans le répertoire partagé spécifié. Les autres applications auront accès à ce fichier et pourront alors accéder aux tables.

Les conditions d'accès sont les suivantes :

- Accès complet en lecture.
- Si un utilisateur entreprend une modification sur un enregistrement, un verrou est placé automatiquement par l'application sur cet enregistrement. De fait une autre application pourra rendre l'enregistrement concerné actif en lecture mais ne pourra pas réaliser d'opération pouvant entraîner une modification.

Si un utilisateur tente de modifier un enregistrement verrouillé, ses entrées au clavier ne sont pas prises en compte et un message d'erreur indiquant que l'enregistrement est verrouillé apparaît. Le message précise quel est l'utilisateur ayant posé le verrou.

POSSIBILITÉS AVANCÉES

Mise en place de mot de passe

Il est possible, à partir de BDE, lors de la création d'une table ou lors d'une restructuration, d'imposer l'utilisation d'un mot de passe pour accéder à une table. Une boite de dialogue demandant d'entrer le mot de passe sera affichée dès que la table sera ouverte.

Sur le réseau, cette table restera accessible par mot de passe. Cependant il est possible d'éviter de passer par la phase d'authentification si on utilise la méthode AddPassword() de l'objet Session.

La méthode AddPassword () sert normalement à ajouter un nouveau mot de passe au composant TSession courant dans le cas où des tables Paradox sont utilisées. Lorsqu'une application ouvre une table Paradox qui nécessite un mot de passe, l'utilisateur doit entrer celui-ci, sauf si un mot de passe valide pour la table a été défini dans Session.

Utilisation d'une table en mode exclusif

Dans certains cas il est souhaitable qu'un seul utilisateur puisse accéder à une table à la fois (que ce soit en lecture ou en écriture).

Pour qu'une table soit utilisée en mode exclusif, il faut positionner à True la propriété Exclusive du composant TTable. La tentative d'accès à la table par un autre utilisateur provoque une interruption qui provoque l'arrêt de l'application si aucun gestionnaire d'interruption a été écrit.

Rafraîchissement des données

Pour être sûr que les données affichées soient bien les données actualisées, il faut exécuter la méthode Refresh () du composant TTable après chaque modification. De cette manière les différents clients de l'application pourront "voir" la dernière version des données.

Charge réseau

Le fait d'utiliser le mécanisme du partage de fichier pour assurer l'accès par le réseau à une base de données est relativement simple à mettre en oeuvre. Il est cependant très pénalisant en terme de charge réseau. Il faut en effet comprendre que dans ce mécanisme l'essentiel du traitement de déroule sur la machine cliente qui charge la totalité de la table, à travers le réseau, pour agir sur elle.

De ce fait le partage du répertoire consomme beaucoup de ressources système et, lorsqu'on réalise le chargement de la table, c'est la totalité de celle-ci (en tous cas par tampons assez importants) qui est transférée depuis la machine de stockage à la machine d'exécution. Si la table est importante et si elle est souvent utilisée, la charge résultante peut devenir excessive.

UTILISATION DE PILOTES ODBC

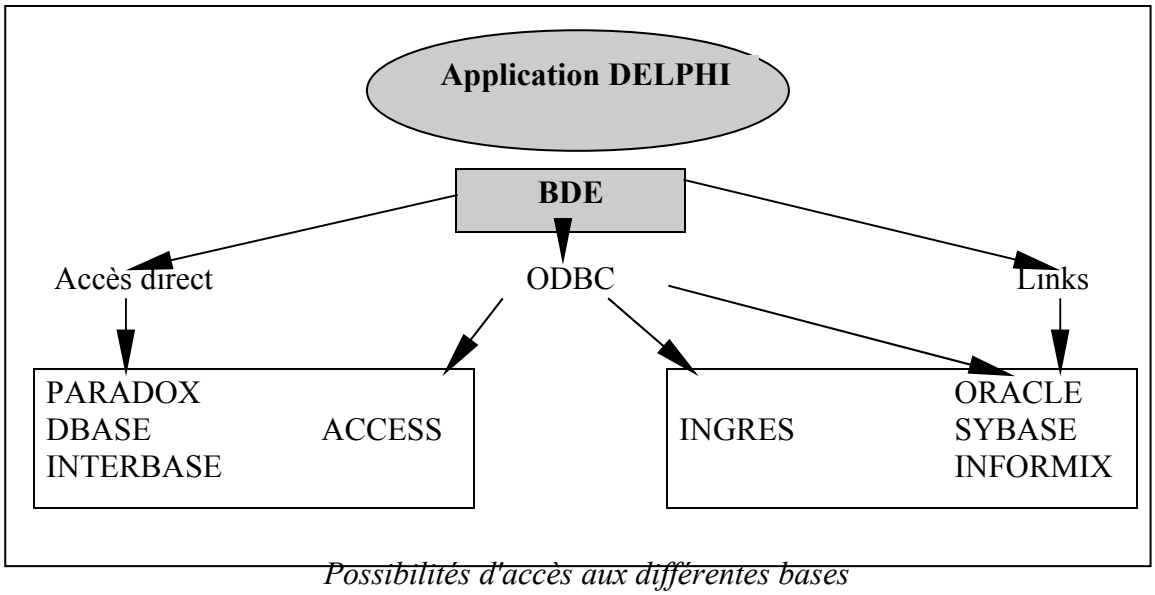
La méthode qui consiste à accéder aux données en exécutant un simple partage de fichiers à ses limites :

- Le traitement est intégralement réalisé sur la machine cliente ;
- La charge réseau est importante.

De plus cette méthode ne peut être utilisée que pour accéder à des données stockées dans des tables dont le format interne est reconnu par BDE (soit les tables aux formats dBase, Paradox et Interbase). Dans ce cas là il n'y a (pratiquement) pas de problème.

Mais, dans un environnement de plus en plus hétérogène on peut être amené à devoir créer une application qui devra pouvoir accéder à des données utilisant un autre format, inconnu de BDE.

Ce peut être des tables créées par un autre SGBD/R micro (tels Access ou FoxPro) ou plus sérieusement des tables gérées par un Serveur SQL (tels Oracle, Sybase, Informix ou Ingres). Si l'on dispose de la version Client / serveur de DELPHI, on peut alors utiliser, dans certains cas, des pilotes spécifiques (links) permettant d'accéder rapidement aux tables concernées. Dans les autres cas il faudra utiliser les services du gestionnaire ODBC proposé par Microsoft.



Possibilités d'accès aux différentes bases

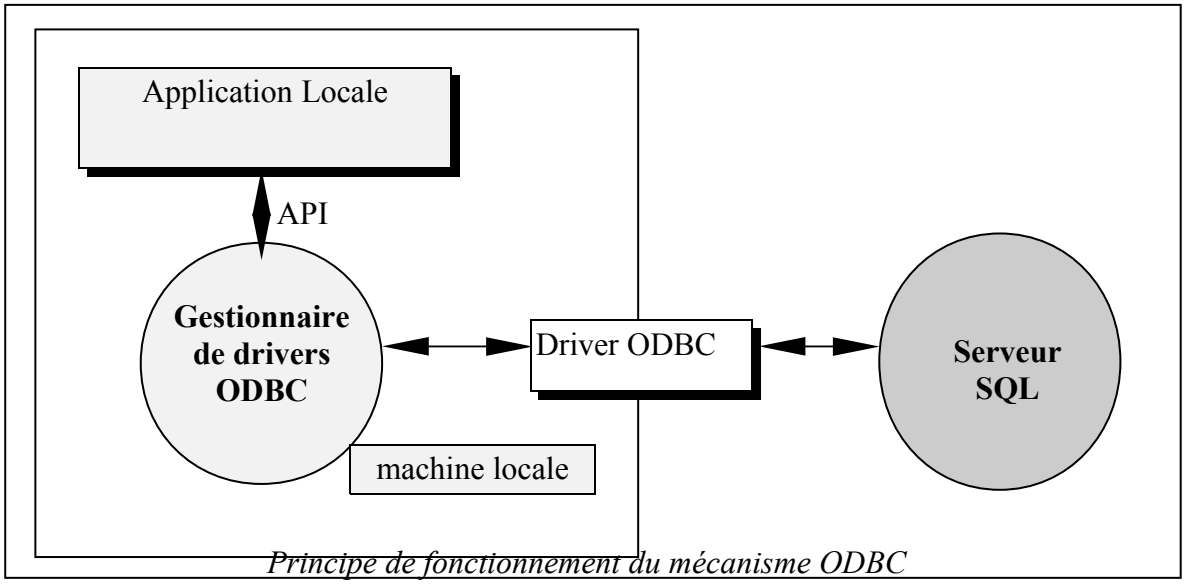
QU'EST-CE QU'UN PILOTE ODBC ?

ODBC est un standard proposé par Microsoft permettant l'accès aux différents SGBD/R et fonctionnant sous Windows.

C'est un ensemble de logiciels, API et pilotes, qui donne aux programmeurs la possibilité d'accéder à des bases de données externes. Il fournit aux éditeurs de bases de données des pilotes adaptés à leurs produits. ODBC repose sur les standards de SQL.

Le mécanisme est constitué essentiellement de:

- Une API permettant d'invoquer le mécanisme à partir d'une application, via une requête SQL.
- Un 'gestionnaire de drivers' (fourni par Microsoft et configurable directement à partir de Windows). Il charge les pilotes nécessaires, oriente les appels vers le bon pilote, gère les messages d'erreurs.
- Le driver ODBC lui-même : il est spécifique au SGBD/R auquel on souhaite accéder et transforme la requête SQL en un format compréhensible par le SGBD/R. Il retourne le résultat. Ce driver est conçu et livré par le fournisseur du SGBD/R.



Principe de fonctionnement du mécanisme ODBC

CONFIGURATIONS INITIALES

Avant de pouvoir utiliser un pilote ODBC il faut réaliser diverses opérations de configuration.

Il va falloir en effet :

- Rechercher et installer un pilote ODBC ;
- Le faire reconnaître par le gestionnaire de pilotes ODBC de Windows ;
- Le configurer pour une base précise ;
- Configurer l'application DELPHI pour qu'elle reconnaisse ce pilote.

Recherche du pilote ODBC

Le premier problème auquel on se heurte consiste à disposer du "bon" pilote ODBC.

Il faut en effet :

- Disposer du pilote prévu pour le type de base de données à laquelle on veut accéder. Ceci n'est pas toujours évident ; s'il est relativement aisé d'obtenir un pilote pour un SGBD/R connu (Oracle, Informix ou Access, ...) il est souvent plus délicat d'obtenir ceux permettant l'accès à des SGBD/R dont les éditeurs sont nettement moins "ouverts" (Ingres, Progress, ...) ;

- Disposer de la bonne version du pilote : Il est relativement difficile de programmer un pilote ODBC. De ce fait les éditeurs ont parfois du mal à fournir un pilote optimisé pour accéder à la dernière version de leur SGBD/R. Si on dispose d'un pilote il faut donc tenter de s'assurer que celui-ci correspond bien à la version du SGBD/R utilisé : il ne doit pas être trop ancien ... mais il ne doit pas être trop récent non plus.

Reconnaissance du pilote ODBC par le gestionnaire ODBC de Windows

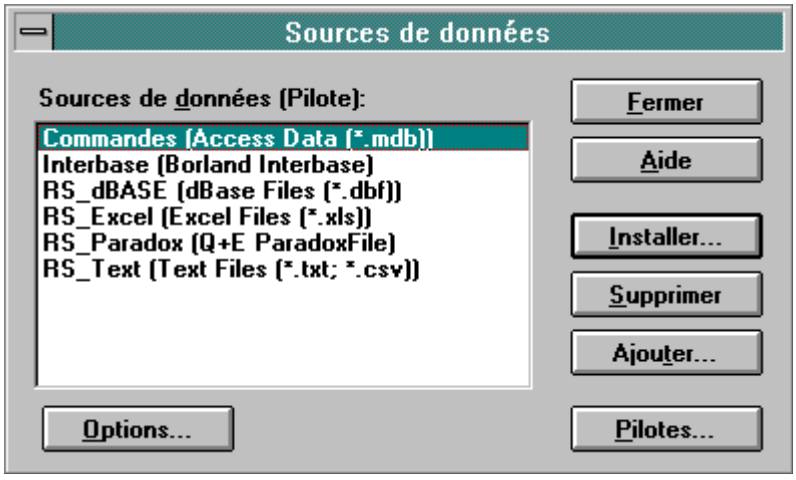
Une fois le pilote récupéré, il faut l'installer et faire en sorte qu'il soit pris en compte par le gestionnaire de pilotes ODBC qui est fourni avec Windows.

Pour cela il faut ouvrir le Panneau de Configuration de Windows et activer l'icône ODBC.

Penser à ouvrir en parallèle l'aide en ligne ODBCINST.HLP, contenue dans le répertoire Windows / System. Ce ne sera pas superflu.

Une boite de dialogue intitulée ' Sources de données' apparaît. Elle permet la gestion des sources de données répertoriées. Une source de donnée correspond à l'ensemble des données auxquelles on souhaite accéder ainsi que les informations nécessaires pour y parvenir.

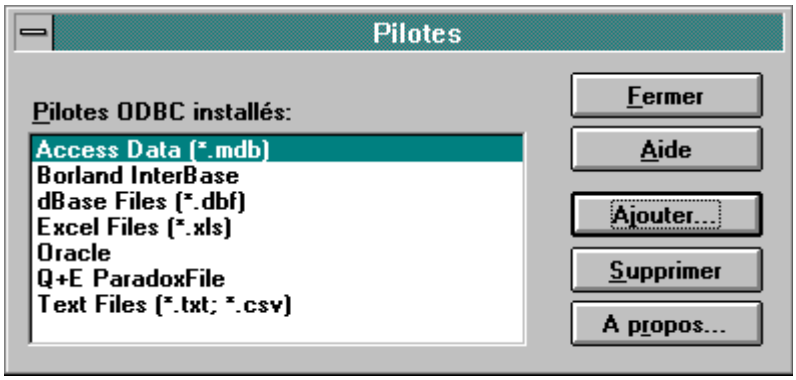
La boite permet d'installer ou de supprimer une source de données. Elle permet aussi d'obtenir des renseignements sur les sources répertoriées et d'installer de nouveaux pilotes.



La boite de dialogue 'Sources de données'

Pour installer un nouveau pilote il faut sélectionner le bouton 'Pilotes ...'

Une nouvelle boite de dialogue apparaît : elle permet la gestion des différents pilotes ODBC.



La boite de dialogue des pilotes ODBC

Le fait de cliquer sur le bouton 'Ajouter...' affiche une boite de dialogue demandant de spécifier le chemin d'accès au pilote (éventuellement via le réseau).

Une fois les renseignements requis fournis, la procédure d'installation du pilote sur le disque est alors démarrée. Son entrée est rajoutée dans la liste.

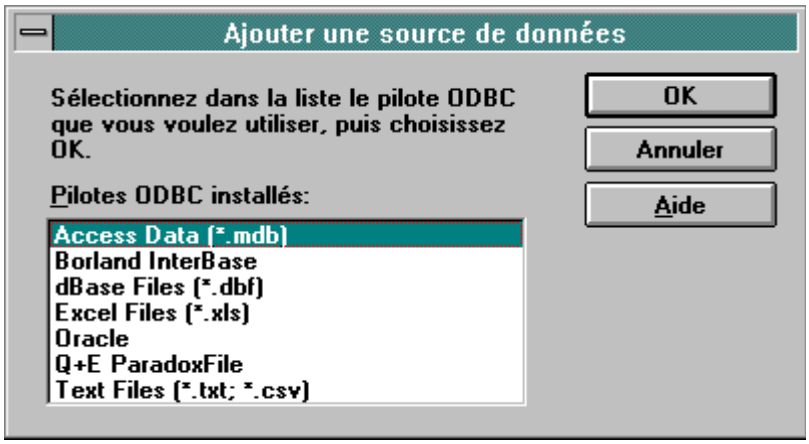
On peut alors revenir dans la boite de dialogue initiale 'Sources de données'.

Configuration de la source de donnée

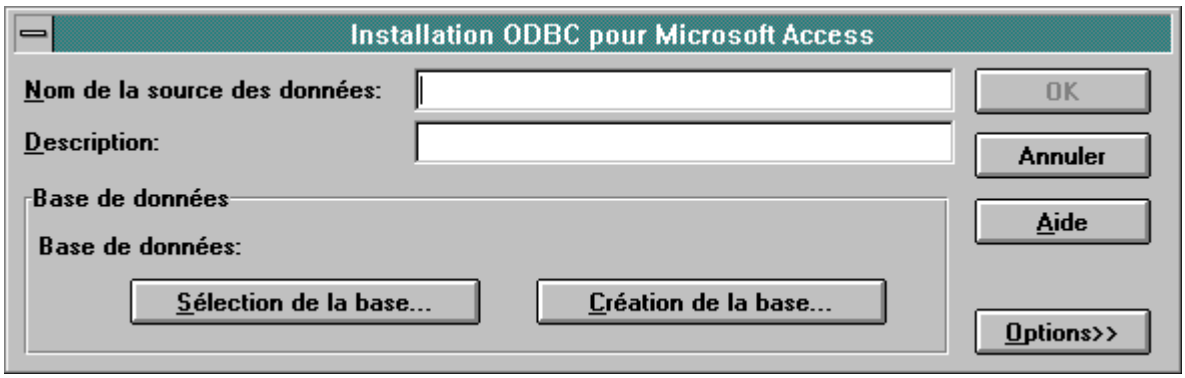
Un même pilote ODBC peut être utilisé pour accéder, à partir de plusieurs applications, à plusieurs bases de données. Pour une application précise, il faut donc indiquer quelle est la base "cible" et indiquer comment y accéder. L'ensemble des informations nécessaires constitue une "source de données".

Il faut, dans un premier temps créer une nouvelle source de données. Pour cela il faut activer le bouton 'Ajouter...' de la boite de dialogue.

Une boite de dialogue permet de définir quel sera le pilote ODBC utiliser pour accéder à cette base de données.



Le fait de choisir le pilote ODBC à utiliser (par exemple : Access) ouvre une nouvelle boite de dialogue qui permet de spécifier la base de données "cible".



Il s'agit alors de définir la source de donnée :

- Il faut lui attribuer un nom (qui peut être différent de celui du ou des fichiers constituant la base). Par exemple on peut indiquer : Gestion des stages.

- Il faut indiquer le chemin d'accès à la base. Ce qui veut dire qu'il faut indiquer, dans la majorité des cas, le lecteur partagé dans laquelle la base est stockée. Ce qui signifie que plus tard, à l'exécution, il faudra, comme dans le cas du fichier partagé, se connecter au préalable à la base de données distante avant de lancer l'application.

Il y a une fenêtre d'installation ODBC par type de base cible. Chacune présente ses propres spécificités.

Par exemple, pour Access, un bouton Options permet de définir la base de données système dans laquelle sont stockées les informations permettant l'authentification des différents utilisateurs.

Il faut en général se reporter à la documentation du SGBD/R cible pour configurer finement la source de données.

La source de données étant configurée, elle est maintenant reconnue par le gestionnaire de drivers ODBC. On peut quitter le Panneau de Configuration.

Configuration au niveau de DELPHI

Il faut maintenant faire reconnaître les configurations effectuées par DELPHI. Pour cela il faut lancer l'utilitaire Configuration Database Engine.

Création d'un nouveau pilote ODBC

Il faut activer le bouton 'Nouveau Pilote ODBC'. Une boite de dialogue apparaît.

Il faut renseigner les différentes zones de saisie comme suit :

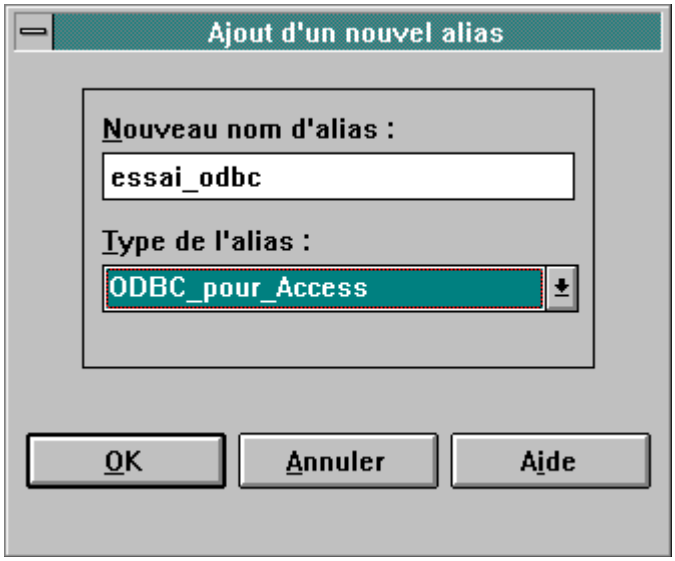
- Donner un nom permettant d'identifier le pilote dans la zone associée à 'Pilote SQL LINKS'. Le configurateur ajoute automatiquement le préfixe 'ODBC_' à ce nom.
- Sélectionner le pilote ODBC par défaut (correspondant au type de SGBD/R cible).
- Sélectionner dans la liste des sources de données proposée celle qu'il faut utiliser.

Au sortir de cette boite de dialogue le nouveau pilote ODBC est référencé.

Création d'un alias

En restant dans le configurateur, il faut créer un nouvel alias. Pour ce faire, il faut activer l'onglet correspondant et choisir 'Nouvel Alias'.

Dans la boite de dialogue qui apparaît, indiquer le nom de l'alias puis, dans la liste déroulante associée à 'Type de l'alias' choisir le nom du pilote ODBC souhaité.



Création d'une application utilisant un pilote ODBC

Une fois toutes les configurations préalables réalisées, il ne reste plus qu'à sortir de l'utilitaire et concevoir l'application cliente en utilisant l'alias créé pour accéder à la base souhaitée.

> Lorsque le pilote ODBC est utilisé pour accéder à un serveur SQL, c'est un composant TQuery, permettant de gérer une requête SQL, qui doit être utilisé en lieu et place d'un composant TTable.

REMARQUES SUR ODBC

Avant d'utiliser le mécanisme ODBC, il y a lieu de bien comprendre la différence qu'il y a entre un pilote ODBC accédant à une base "micro" (Paradox, Access, FoxpPro, etc...) et un pilote accédant à un serveur SQL.

Dans le premier cas, l'application accède aux tables "inertes" stockées sur le disque (le pilote ODBC se contentant de rendre le format de stockage des données lisible). De ce fait, comme dans le cas d'un partage de fichier, tout le traitement est réalisé par l'application cliente, ce qui se traduit par une grande charge réseau et des performances globales peu satisfaisantes.

Dans le deuxième cas, le pilote ODBC permet d'accéder directement au serveur SQL via des requêtes adéquates. La charge réseau est nettement moindre et l'application cliente obéit aux règles d'administration du serveur SQL. Même si les performances globales sont moindres que si l'on utilise un pilote natif elle restent néanmoins satisfaisantes.

LES + DE DELPHI 3

ODBC : UNE COUCHE D'ABSTRACTION SUPPLÉMENTAIRE

Le moteur de bases de données Borland peut communiquer avec un grand nombre de systèmes de bases de données. Cette caractéristique est intéressante si le développeur veut changer de système de gestion de bases de données. Pour communiquer avec n'importe quel type de système de gestion de bases de données, il existe ODBC.

ODBC signifie Open Database Connectivity. ODBC vous permet de communiquer virtuellement avec n'importe quel SGBD (Système de Gestion de Bases de données) grâce à une interface générique appelée pilote ODBC. Le pilote ODBC assure l'interfaçage avec un SGBD particulier en normalisant les accès à travers un ensemble d'appels à une API.

L'utilisation d'ODBC garantit à l'application un maximum de souplesses, puisqu'il suffit en théorie de changer de pilote ODBC pour changer de SGBD. En réalité, il y a quelques différences d'un SGBD à l'autre. Si la vitesse d'exécution pose problème, il est possible

d'accéder directement à l'API ODBC. Cependant, cela introduit un nouveau niveau de difficulté.

DANS QUELLES CIRCONSTANCES UTILISER ODBC ?

Il arrive souvent que le développeur ne sache pas à quelle échelle développer sa base (locale, à fichiers partagés ou Client/Serveur). ODBC permet à ce développeur de créer une base de données locale, et de la transformer selon le modèle Client/Serveur avec un minimum de modifications.

Il pourra également arriver que le développeur soit obligé de tenir compte d'impératifs de compatibilité. Si par exemple une application de paie utilise SyBase SQL, Delphi pourra utiliser ODBC pour se connecter à la base de données.

Dans la terminologie ODBC, un pilote ODBC est une libraire qui sait communiquer avec le SGBD sous-jacent. Les pilotes ODBC sont souvent fournis par le vendeur du SGBD. Une source de données ODBC est une instance d'ODBC utilisant un pilote particulier. Par exemple, vous pouvez disposer d'un pilote ODBC Sybase SQL server, et RHDATA est une source de données ODBC qui est un serveur SQL des données relatives aux ressources humaines. Il peut y avoir plusieurs sources de données utilisant le même pilote, par exemple INVENT pointerait vers les informations d'inventaire.

L'utilitaire ODBCAD32 sert à créer et configurer des sources de données. Pour chaque source de données les informations sont distinctes. Par exemple, une source de données ODBC Microsoft Access doit connaître l'emplacement du fichier .mdb, alors qu'une source de données SQL devra comporter l'adresse du serveur et l'interface réseau nécessaire pour y accéder.

Pour utiliser un pilote ODBC et communiquer grâce à ODBC sous Delphi, vous devez créer un nouveau moteur de bases de données. Cela peut sembler étrange, mais vous devez créer un pilote ODBC vers le SGBD voulu, et un moteur de bases de données Borland vers le pilote ODBC. Pour créer un moteur de bases de données Borland, procédez ainsi :

1. Lancez l'utilitaire de configuration de bases de données Borland depuis le menu démarrer de Windows 95 ou NT.
2. Cliquez sur l'onglet Configuration. Sélectionnez Configuration|Pilotes|ODBC. Faites un clic droit sur ODBC, et sélectionnez Nouveau dans le menu contextuel.
3. Mettez le nom du pilote SQL Link à ODBC_XXXX, où XXXX est le nom du pilote Borland que vous voulez créer (voir Figure 11.12).
4. Sélectionnez le pilote ODBC à utiliser pour accéder aux données ainsi qu'une source de données afin de créer un alias ODBC, puis cliquez sur OK. Sélectionnez Objet|Quitter pour fermer l'Administrateur BDE.

Voilà tout pour la création du pilote. Pour accéder aux données, le meilleur moyen est de créer un alias en utilisant le Module base de données.

BASE DE DONNÉES MICROSOFT ACCESS AVEC ODBC

Définissez la source de données ODBC de la façon suivante :

1. Lancez ODBCAD32.
2. Cliquez sur Ajouter dans la boîte de dialogue Sources de données. La liste des pilotes ODBC installés s'affiche.
3. Choisissez *.mdb puis cliquez sur Terminer.
4. Donnez un nom et une description à la source de données Access. Dans cet exemple, nous mettrons **TestAccess.**
5. Cliquez sur Choisir. Vous voyez apparaître une liste de fichiers .mdb.
6. Choisissez le fichier qui vous intéresse.
7. Cliquez sur OK. Cliquez de nouveau sur OK pour sortir du programme ODBCAD32.

Avant de pouvoir utiliser le pilote ODBC que vous venez de définir, vous devez exécuter l'Administrateur BDE (il fait partie des programmes installés dans le menu Démarrer). La nouvelle source ODBC sera alors automatiquement prise en compte. Quittez l'Administrateur pour enregistrer les modifications.

Pour que la prise en compte des nouveaux pilotes ODBC se fasse automatiquement, il faut que le paramètre AUTO ODBC de l'Administrateur BDE soit à TRUE. Vous pouvez vérifier ce paramétrage de la façon suivante :

1. Exécutez l'Administrateur BDE.
2. Sélectionnez l'onglet Configuration.
3. Sélectionnez Configuration|Système|INIT.
4. La Définition de INIT apparaît alors dans le volet de droite. La propriété AUTO ODBC est sur la première ligne.

Pour utiliser l'alias ODBC sous Delphi, procédez comme ainsi :

1. Créez un nouveau projet, et ajoutez un composant TTable.
2. Mettez la propriété DatabaseName à TstAccess.
3. Si une boîte de dialogue demandant un nom d'utilisateur et un mot de passe apparaît, cliquez sur OK.
4. Notez que les tables de la base de données Access sont maintenant disponibles dans la propriété TableName. Choisissez une table, et continuez comme avec n'importe quelle table Delphi.

Si, lorsque vous mettez à TRUE la propriété Active *d'un composant* TTable*, vous obtenez le message "objet non trouvé', c'est que vous avez oublié d'exécuter l'Administrateur BDE afin de mettre à jour la configuration BDE.*

SÉCURITÉ, LES MOTS DE PASSE

Vous avez entendu parler des pirates informatiques qui récupèrent des données top secret en décodant des informations repiquées sur une ligne de téléphone à l'aide d'un Cray 42. Il y a certes un peu d'exagération dans tout cela, mais la sécurité est un problème important qu'il ne faut pas négliger. La sécurité des données recouvre l'ensemble des opérations permettant à certains utilisateurs ou groupes d'utilisateurs d'accéder à des données en lecture ou en édition.

Dans l'exemple des nombres premiers, que se passerait-il si n'importe qui pouvait accéder directement à la base de données (sans passer par l'application Delphi que vous avez amoureusement peaufinée) pour aller ajouter le nombre 42 ou 513 (513 non plus n'est pas premier) ?

LES NIVEAUX DE SÉCURITÉ

Différents niveaux de sécurité sont disponibles selon le SGBD utilisé. Nous nous limiterons au cas des utilisateurs individuels, laissant de côté les groupes et les autres applications (les fonctionnalités sont identiques). Les niveaux de sécurité les plus courants sont :

- Aucun — Toute personne ayant physiquement accès aux données peut en faire ce qu'elle veut.
- Tout ou rien — Une fois que l'utilisateur a reçu le droit d'accéder à la base de données, il est libre d'y faire ce qu'il lui plaît.
- Multi-utilisateur, au niveau des tables — Chaque utilisateur peut accéder à des table différentes dans différents modes. Ainsi, un administrateur pourra éditer ou modifier des enregistrement,tandis qu'un utilisateur sans privilèges ne pourra que consulter la table.
- Multi-utilisateur, au niveau des champs — Ce système est le plus souple : l'accès peut être défini, utilisateur par utilisateur, table par table, et champ par champ. Il est par exemple possible de ne voir que certains champs d'une table, et parmi ces champs avoir quelques droits de modification, mais pas d'effacement.

AUTHENTIFICATION

Maintenant que vous savez définir des privilèges, utilisateur par utilisateur, le problème principal est de savoir que tel ou tel utilisateur tente d'accéder à la base. Pour vérifier que l'utili-sateur est bien qui il prétend être, on a recours à l'authentification. Quand un utilisateur a prouvé son identité, on dit qu'il est authentifié (ou certifié).

Le moteur de bases de données doit également savoir authentifier un utilisateur et déterminer son type d'accès. La façon de procéder la plus courante repose sur des mots de passe. Paradox utilise un mot de passe maître qui définit ce propriétaire. Des mots de passe secondaires peuvent être définis par ce propriétaire, et il est possible de leur attribuer des accès par champ. Une autre méthode consiste à authentifier les utilisateurs par leur nom et un mot de passe. Cette méthode est meilleure car elle repose sur le nom de l'utilisateur et non sur un mot de

passe. D'un point de vue administratif, le niveau de sécurité est plus satisfaisant puisque le propriétaire de la base n'a plus à connaître le mot de passe de chaque utilisateur.

LES THREADS

En complément de son noyau multitâche, l'API Win32 propose le multitraitement. Cela permet à un processus de lancer plusieurs traitements simultanément. Chaque traitement pourra effectuer une tâche précise.

Cette notion est plus forte que la gestion classique des processus sous UNIX, à l'aide de la commande fork.

Ici, il s'agit d'une fonction qui s'exécutera de manière indépendante des autres fonctions du processus principal. Lorsque le code se termine, le traitement prend fin, on ne revient pas dans le code de la procedure appelante.

Microsoft désigne ce traitement sous le nom de thread, et on dit que l'API Win32 permet le multithreading.

L'ordonnanceur de threads

Les threads représentent les traitements effectués par un processus. Un processus doit au moins exécuter un traitement pour exister dans le système. Pour lui, les threads sont vus comme des programmes indépendants et il les gère comme les processus. Une liste circulaire de threads permet alors au système de donner régulièrement la main à un traitement. Les threads sont ordonnancés comme les processus et prennent les mêmes états, à savoir actif, en sommeil, …

Les manipulations possibles sur un threads sont les suivantes :

- Création
- Changement d'état
- Arrêt / démarrage
- Destruction

Endormir et réveiller un thread

Pour suspendre un thread, il suffit d'appeler la fonction SuspendThread avec, comme paramètre, le handle relatif du thread à suspendre. Le thread passe alors à l'état SUSPEND.

Function SuspendThread (hThread : Thandle) : Dword ; stdcall ;

Pour le réveiller par la suite, il suffit d'appeler la fonction ResumeThread en lui ^passant le handle du thread endormi.

Function ResumeThread (hThread : Thandle) : Dword ; stdcall ;

Le système gère un compteur pour l'état endormi d'un thread. Cela signifie que les appels successifs de la fonction SuspendThread sont comptabilisés pour un thread donné. Il faudra

alors mettre autant d'appel à ResumeThread pour que le thread se réveille effectivement. En revanche, les appels successifs à la routine ResumeThread ne sont pas cumulatifs.

Vous pouvez aussi endormir un thread pour un temps infini avec la fonction Sleep. Normalement vous spécifiez un temps en millièmes de secondes, mais si vous donnez pour temps la constante INFINITE, le thread est préempté par le système.

Procedure Sleep (dwMilliseconds : DWORD) ; stdcall ;

Par exemple, pour endormir un thread durant une minute, voilà ce qu'il convient de faire :

```
Procedure EndortThread (hThread : Thandle) ;
Begin
        SuspendThread (hThread) ;
        Sleep(1000) ;
        ResumeThread(hThread) ;
end ;
```

LA CREATION D'UN THREAD

La création d'un thread s'effectue par la fonction CreateThread. Le premier thread d'un programme est créé par le système et porte le nom de thread principal. Sans ce premier thread, l'application ne pourrait s'exécuter.

le prototype de la fonction CreateThread est le suivant :

```
function CreateThread(      lpThreadAttributes: Pointer;
                            dwStackSize: DWORD;
                            lpStartAddress: TFNThreadStartRoutine;
                            lpParameter: Pointer;
                            dwCreationFlags: DWORD;
                            var lpThreadId: DWORD): THandle; stdcall;
```

Paramètre	Description
LpThreadAttributes	Permet de définir les privilèges de sécurité du thread
DwStackSize	Détermine la taille de la pile du thread. La valeur 0 indique une pile standard
LpStartAddress	Pointeur sur la fonction de traitement. La fonction doit être qualifiée par StdCall
LpParameter	Paramètre éventuel passé à la fonction de traitement
DwCreationFlags	Drapeau de création du thread. Les valeurs possibles sont CREATE, SUSPEND ou 0
LpThreadId	Variable de type Dword recevant l'ID du thread créé

L'adresse de la fonction de traitement passée en paramètre devra être une procedure sans paramètre. Le type TFNThreadStartRoutine est défini comme suit :

TFNThreadStartRoutine = TFarProc;

Exemple :

Supposons qu'il faille incrémenter un compteur dans une zone d'édition parallèlement à l'exécution de la tache principale. Plutôt que d'utiliser un composant Timer, il est possible d'utiliser un thread dont le code est :

```
Procedure mon_traitement (edit : tedit) ; stdcall ;
Var
        I : integer ;
Begin
        For i := 0 to 32767 do
                Edit.text :=inttostr(i) ;
end ;
```

On placera un bouton dont le but est de créer le thread :

```
Procedure Tform1.Button1Click (Sender : Tobject) ;
Begin
        ThreadHdle := CreateThread (nil, 0, @montraitement, edit1, 0, ThreadID) ;
End ;
```

Le code du thread généré s'exécute simultanément avec le thread principal de l'application. Puisque le thread principal n'est pas occupé, il est possible de manipuler la fenêtre, ainsi que les autres options du programme.

La mémoire d'un thread

Chaque thread possède sa propre pile et ses propres registres nommés CONTEXT du thread. Il peut aussi accéder à toutes les variables globales du processus propriétaire (voir fichier Windows.pas).

L'arrêt d'un thread

Pour terminer l'exécution d'un thread, vous disposez de 2 routines ExitThread et TerminatedThread

La fonction ExitThread permet à un thread de mettre fin à sa propre exécution. Cette fonction doit recevoir en paramètre le code de sortie du thread.

procedure ExitThread(dwExitCode: DWORD); stdcall;

La deuxième fonction permet à un processus de mettre fin à un de ses threads.

function TerminateThread(hThread: THandle; dwExitCode: DWORD): BOOL; stdcall;

Ainsi, les appels suivants sont équivalents :

```
procedure Tform1.ButtonClick( Sender : Tobject) ;
begin
        TerminateThread (GetCurrentThread, 0) ;
End ;

procedure Tform1.ButtonClick( Sender : Tobject) ;
begin
        ExitThread ;
End ;
```

L'utilisation de TerminateThread est déconseillée car les DLL ne sont pas averties de la fin d'un thread par cet appel. De plus, la pile du thread n'est pas libérée ^par cette fonction. Elle ne doit être utilisée que dans les cas critiques.

Une petite gentillesse….

S'endormir volontairement

Vous pouvez être gentil vis à vis des autres threads en vous suspendant volontairement. Pour ce faire, appelez la fonction sleep :

Procedure sleep (dwmilliseconds : DWORD) ; stdcall ;

Attendre un évènement Windows

Vous pouvez aussi faire appel à la fonction WaitMessage pour qu'un thread s'endorme et passe la main à un autre thread jusqu'à ce qu'il reçoive de nouveaux messages.

Function WaitMessage : Bool ; stdcall ;

La classe Tthread

Delphi propose la classe Tthread pour représenter un objet thread. L'utilisation et le développement d'applications deviennent alors plus surs et plus faciles à gérer. Voici la définition de la classe Tthread :

Que dit l'aide en ligne

TThread est une classe abstraite permettant la création de plusieurs threads d'exécution séparée dans une application.

Chaque nouvelle instance d'un objet dérivé de TThread est un nouveau thread d'exécution. Plusieurs instances d'une classe dérivée de TThread, rendent multithread une application Delphi. Pour utiliser des threads dans une application, dérivez une nouvelle classe de TThread et surchargez ses méthodes.

Quand une application est exécutée, elle est chargée en mémoire, prête à s'exécuter. A ce moment, elle devient un processus comportant un ou plusieurs threads contenant les données, le code et d'autres ressources système du programme. Un thread exécute une partie d'une application et se voit attribué du temps CPU par le système d'exploitation. Tous les threads d'un processus partagent le même espace d'adresse et peuvent accéder aux variables globales du processus.

L'utilisation des threads améliore les performances d'une application en :
- *Gérant les entrées de plusieurs périphériques de communication.*
- *En distinguant des tâches de priorités différentes. Par exemple, un thread de priorité élevée gère des tâches critiques alors qu'un thread de faible priorité gère les autres tâches.*

L'utilisation des threads doit se faire en respectant les recommandations suivantes:
- *Conserver la trace d'un trop grand nombre de threads gaspille du temps CPU; sur un système disposant d'un seul processeur, la limite est de 16 threads par processus.*
- *Si plusieurs threads actualisent les mêmes ressources, synchronisez les threads pour éviter les conflits.*
- *Toutes les méthodes accédant à un composant VCL ou actualisant une fiche doivent être appelées dans le thread principal VCL.*

Pour créer et utiliser un nouvel objet thread, il faut:
- *Choisir Fichier | Nouveau | Objet Thread pour créer une nouvelle unité contenant un objet dérivé de la classe TThread.*
- *Définir le constructeur du nouvel objet thread.*
- *Définir la méthode Execute de l'objet thread en insérant le code à exécuter quand le thread est exécuté.*
- *Transmettre tous les appels utilisant un composant VCL à la méthode Synchronize pour que le thread VCL principal exécute l'appel afin d'éviter les conflits multithreads.*

Voir DELPHI 3.0/DEMOS/THREADS pour des exemples de programmes multi-threads.

Les propriétés publiques de la classe TThread

FreeOnTerminate détermine si l'objet thread est automatiquement détruit lorsque le thread s'achève.

property FreeOnTerminate: Boolean;

Description
FreeOnTerminate a la valeur False par défaut. Si cette valeur est conservée, l'objet thread doit être explicitement détruit dans le code de l'application.

La propriété **Handle** contient le handle du thread.

property Handle: THandle;

Description
Utilisez la propriété Handle dans les appels de fonctions de manipulation de thread de l'API Win32.

Priority détermine le niveau de priorité du thread relativement aux autres threads du processus.

type TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical);
property Priority: TThreadPriority;

Description
La propriété Priority est d'un type énuméré dont la valeur par défaut est tpNormal; choisissez, selon les besoins, une priorité plus haute ou plus basse.
Le type TThreadPriority définit les valeurs possibles, énumérées dans le tableau suivant, de la propriété Priority du composant TThread. Windows répartit les cycles de la CPU à chaque thread en se fondant sur les priorités relatives; la propriété Priority ajuste le niveau de priorité d'un thread plus ou moins haut selon sa position.

Valeur	Signification
tpIdle	Le thread ne s'exécute que lorsque le système est inoccupé. Windows n'interrompt pas les autres threads pour exécuter un thread de priorité tpIdle.
tpLowest	La priorité du thread est deux points en dessous de la normale.
tpLower	La priorité du thread est un point en dessous de la normale.
tpNormal	La priorité du thread est normale.
tpHigher	La priorité du thread est un point au dessus de la normale
tpHighest	La priorité du thread est deux points au dessus de la normale.
tpTimeCritical	Le thread a la priorité la plus élevée.

Attention …
"Gonfler" la priorité du thread pour une opération utilisant intensivement la CPU peut "sous-alimenter" les autres threads de l'application. Il ne faut accorder une priorité élevée qu'à des threads qui passent l'essentiel du temps à attendre des événements extérieurs.

La propriété **ReturnValue** est l'équivalent pour un thread de la variable Result d'une fonction.

property ReturnValue: Integer;

Description
La propriété ReturnValue permet d'indiquer, à l'application ou aux autres threads, la réussite, l'échec ou le résultat numérique du thread. La méthode WaitFor renvoie la valeur stockée dans ReturnValue.

La propriété **Suspended** indique si un thread est suspendu.

property Suspended: Boolean;

Description
Les threads suspendus ne sont pas exécutés jusqu'à ce qu'il soient relancés. Affectez la valeur True à Suspended pour suspendre un thread et lui affecter la valeur False pour le relancer.

La propriété **Terminated** indique s'il a été demandé au thread de s'arrêter. La méthode Terminate affecte la valeur True à Terminated.

property Terminated: Boolean;

Description
La méthode Execute et toutes les méthodes appelées par Execute doivent tester périodiquement Terminated et sortir si sa valeur est True.

La méthode Terminate constitue le moyen correct d'arrêter l'exécution d'un thread, mais elle suppose la coopération du code de la méthode Execute du thread. Il est préférable d'utiliser Terminate plutôt que l'appel de la fonction TerminateThread de l'API Win32.

La propriété **ThreadID** est un identificateur de thread identifiant le thread de manière unique dans le système. ThreadID est différent du handle contenu dans la propriété Handle.

property ThreadID: THandle;

Description
ThreadID est utile dans les appels des fonctions de manipulation de thread de l'API Win32.

Les méthodes publiques de la classe TTHread

La méthode **Create** crée une instance d'un objet thread.

constructor Create(CreateSuspended: Boolean);

Description
Si CreateSuspended a la valeur False, Execute est appelée immédiatement. Si CreateSuspended a la valeur True, la méthode Execute n'est appelée qu'après un appel de Resume (on crée un thread endormi).

La méthode **Destroy** détruit l'objet thread et libère la mémoire qui lui est allouée.

destructor Destroy; override;

Description
Destroy signale au thread de s'arrêter et attend l'arrêt du thread avant d'appeler le destructeur hérité Destroy.Il permet de libérer la mémoire occupée par un objet. Comme pour les autres objets de DELPHI, vous n'appellerez pas directement le destructeur mais plutôt la méthode free, qui est plus sure.
Le problème se pose de savoir si le thread a terminé son travail. Cette information peut être transmise par l'évènement OnTerminate, qui est déclenché quand le thread a terminé son exécution. De plus, la propriété FreeOnTerminate de type booléen permet d'indiquer à l'objet thread de libérer sa mémoire en fin d'exécution du traitement.

Execute est une méthode abstraite devant être surchargée; elle contient le code à exécuter quand le thread commence.

procedure Execute; virtual; abstract;

Description
Un thread commence lorsque Create est appelée avec le paramètre CreateSuspended initialisé à False ou si la méthode Resume est appelée après un appel de Create dans lequel CreateSuspended est initialisé à True. Surchargez cette méthode et insérez le code à exécuter quand le thread commence. Execute doit tester la valeur de la propriété Terminated afin de déterminer s'il faut sortir du thread.

Remarque
N'utilisez pas les propriétés et méthodes d'autres objets directement dans la méthode Execute d'un thread. Il faut séparer l'utilisation des autres objets dans un appel de procédure distinct et appeler cette procédure en la transmettant comme paramètre à la méthode Synchronize.

La méthode **Resume** reprend l'exécution d'un thread interrompu.

procedure Resume;

Description
Il est possible d'imbriquer les appels de la méthode Suspend; Resume doit alors être appelée le même nombre de fois que Suspend avant que l'exécution du thread reprenne.

La méthode **Suspend** interrompt l'exécution du thread.

procedure Suspend;

Description
Appelez Resume pour interrompre l'exécution. Il est possible d'imbriquer les appels de la méthode Suspend; Resume doit alors être appelée le même nombre de fois que Suspend avant que l'exécution du thread reprenne.

Exemple :
Voici une méthode permettant à un traitement de s'effectuer toutes les secondes

```
Procedure Tform1.Button1Click (Sender : Tobject) ;
Begin
        MonThread := TmonThread.create (true) ;
        While true do
        Begin
                MonThread.Resume ;
                Sleep(1000) ;
                MonThread.Suspend ;
                Sleep(1000) ;
        End ;
End ;
```

Terminate signale au thread de s'arrêter en affectant la valeur True à la propriété Terminated.

procedure Terminate;

Description
La méthode Execute du thread et toutes les méthodes appelées par Execute doivent tester régulièrement Terminated et sortir si elle a la valeur True.

Remarque
Le traitement n'est pas interrompu par cette méthode.

La méthode **WaitFor** attend la fin du thread, puis renvoie la valeur de la propriété ReturnValue.

function WaitFor: Integer;

Description
WaitFor ne renvoie de valeur qu'à l'arrêt du thread. Le thread doit donc être arrêté soit par la méthode Execute, soit parce que la propriété Terminated a la valeur True. N'appelez pas WaitFor dans le contexte du thread VCL principal si le thread utilise la méthode Synchronize. Une telle utilisation pourrait provoquer un verrou mortel qui bloque l'application ou déclencher une exception EThread.

Synchronize attend que le thread VCL entre dans la boucle des messages avant d'autoriser l'exécution de la méthode qu'elle tente de synchroniser. Si le thread VCL principal a appelé WaitFor, il n'entre pas dans la boucle des messages et il n'est plus possible de sortir de Synchronize. TThread détecte cette situation et déclenche une exception EThread dans le thread, ce qui provoque son arrêt. Si elle n'est pas bloquée dans la méthode Execute, l'application s'arrête alors. Si Synchronize est déjà en train d'attendre le thread VCL principal lors de l'appel de WaitFor, TThread ne peut intervenir et l'application se bloque.

Les événements de la classe TTHread

L'événement **OnTerminate** est déclenché après le retour de la méthode Execute du thread, mais avant la destruction du thread.

type TNotifyEvent = procedure (Sender: TObject) of object;
property OnTerminate: TNotifyEvent;

Description
Le gestionnaire d'événement OnTerminate est appelé dans le contexte du thread principal VCL. Les propriétés et méthodes VCL sont donc utilisables sans contraintes. L'objet thread peut également être libéré dans le gestionnaire d'événement.
Le type TNotifyEvent est le type, sans paramètre, de ces événements. Ces événements notifient uniquement le composant qu'un événement simple s'est produit, ici l'événement OnTerminate.

Exemple
Si le but du thread est de rechercher un mot dans une liste chaînée de chaînes de caractères, à la fin de la recherche on peut afficher le mot dans une boite de liste comme ceci :

```
Procedure TForm1.Affiche (t : TObject) ;
Begin
        Listbox1.Items.Add ((t as TsearchThread).TraouveA) ;
End ;
```

```
Procedure TForm1.Button1Click (Sender : TObject) ;
Var
        SearchThread : TsearchThread ;
Begin
        SearchThread :=TsearchThread.Create (Memo1.Lines) ;
        SearchThread.FreeOnTerminate :=true ;
        SearchThread.OnTerminate :=Affiche(SearchThread) ;
        SearchThread.Resume ;
End ;
```

Dans cet exemple, la méthode TrouveA renvoie la position de l'élément recherché.

LA SYNCHRONISATION DES THREADS

La synchronisation consiste à mettre en sommeil des threads (en libérant leurs tranches de temps CPU) jusqu'au déclenchement d'un événement donné. Pour cela, il existe deux manières très simples de mettre en sommeil un thread en appelant les routines Win32 WaitForSingleObject et WaitForMultipleObjects. La première routine permet de mettre en sommeil un thread jusqu'à ce qu'un objet soit signalé alors que la seconde permet de mettre en sommeil un thread jusqu'à ce qu'au moins un objet sur plusieurs soit signalé ou jusqu'à ce qu'ils le soient tous.

Function WaitForSingleObject (hHandle : Thandle; dwMilliseconds : DWORD) : DWORD;

Function WaitForMultipleeObjects (nCount : DWORD; lpHandles : PWOHandleArray; bWaitAll : BOOL; dwMilliseconds : DWORD) : DWORD;

Dans la routine ***WaitForSingleObject***, le paramètre hHandle est le handle de l'objet dont vous souhaitez attendre la signalisation. DwMilliseconds est le nombre de millisecondes pendant lesquels vous êtes disposés à attendre (passez la constante INFINITE pour une attente infinie). La valeur de retour est soit la constante WAIT_FAILED, soit WAIT_OBJECT_0.

Dans la routine ***WaitForMultipleeObjects***, le paramètre nCount est le nombre de handles que vous souhaitez scruter et lpHandles est un pointeur vers un tableau de leurs handles. DWaitAll vaut TRUE si vous souhaitez attendre la signalisation de tous les handles ou FALSE si vous souhaitez que l'appel finisse dès que l'un des handles est signalé. La valeur de retour est soit WAIT_FAILED, soit une valeur indiquant quel handle a été signalé. Si bWaitAll vaut TRUE dans l'appel initial et que l'attente n'a pas dépassé la durée du timeout, tous les handles ont étés signalés et vous ne pouvez rien déduire de la valeur de retour.

ANNEXE :PARAMÈTRES DU PILOTE ACCESS

Les applications utilisant le BDE peuvent maintenant ouvrir ou créer des tables utilisant le pilote MSACCESS. Pour travailler avec les tables Access, choisissez MSACCESS comme nom de pilote sur la page Configuration de l'administrateur BDE et mettez en surbrillance le paramètre de configuration désiré ou créez ou sélectionnez un alias sur la page Base de données qui utilise MSACCESS comme nom de pilote. Supprimez l'ancienne valeur et saisissez une nouvelle valeur dans la boîte texte appropriée. Vous ne pouvez modifier que les paramètres sans libellés en gras.

Dans le tableau suivant, Paramètre énumère tous les paramètres définissant le type de pilote sélectionné, et leur valeur par défaut. Lorsque le pilote est installé en premier, toutes les valeurs sont initialisées à celles par défaut. Description donne, de manière brève, l'usage du paramètre en surbrillance.

Paramètre ***Description***

VERSION Le numéro de version interne du pilote Access.

TYPE Type de serveur auquel ce pilote permet la connexion. Ce peut être SERVER (serveur SQL) ou FILE (standard, fichier du serveur).

DLL32 Le nom de la DLL 32 bits du pilote. Par défaut: IDDAO32.DLL

DRIVER FLAGS Indicateur spécifique de production interne. Ne pas modifier sans instruction de la part du support Borland.

TRACE MODE Une valeur numérique (masque de bit) spécifiant les informations d'analyse à enregistrer. L'appel à OutputDebugString de Windows est utilisé pour sortir les informations requises dans la fenêtre de débogage. Le tableau suivant montre les informations enregistrées à partir du masque de bit:

Masque de bit	**Informations enregistrées**
0x0001	instruction de requête préparée
0x0002	instructions de requête exécutées
0x0004	erreurs du distributeur
0x0008	options d'instruction (qui est alloué, libéré)
0x0010	connexion / déconnexion
0x0020	transaction
0x0040	E/S BLOB
0x0080	disparités
0x0100	appels du distributeur

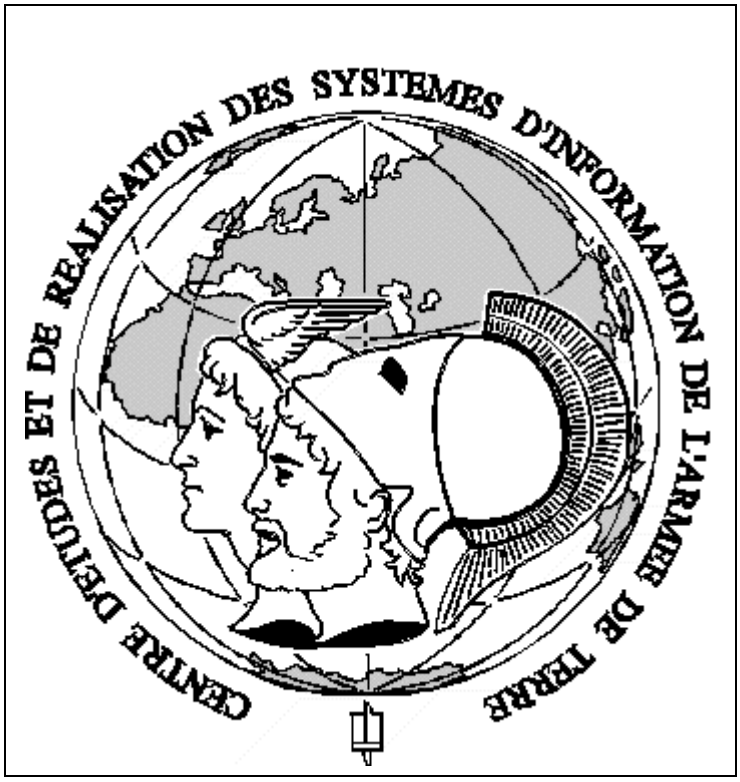
DATABASE NAME Le lecteur, chemin et nom du fichier .MDB auquel vous voulez accéder.

USER NAME Nom par défaut pour accéder à la base de données.

OPEN MODE Mode avec lequel le pilote ouvre la connexion avec la base de données. Ce peut être READ/WRITE ou READ ONLY. Par défaut: READ/WRITE

LANGDRIVER Pilote de langue utilisé pour déterminer l'ordre de tri de la table et le jeu de caractères.

LA CHARTE GRAPHIQUE DU CERSIAT



Introduction générale Charte graphique V1.0

Introduction

La nouvelle interface graphique de Windows NT 4.0 nous conduit à repenser une partie des normes de développement d'applications. Ce nouveau système d'exploitation n'amène pas seulement des nouvelles fonctionnalités, mais également - et surtout - une nouvelle philosophie. En effet, alors que Windows 3.x était orienté *application*, Windows NT 4.0 est lui orienté *document*, l'application n'étant que l'outil nécessaire à son traitement. Certains concepts sont donc à reconsidérer et aujourd'hui encore plus qu'hier, le travail d'*infographiste* devient un véritable métier à part entière. Ce document, en exposant des directives éprouvées en matière d'interface homme - machine (IHM ou GUI en anglais pour *Graphic User Interface*) est appelé à devenir le "Guide de l'infographiste de l'armée de terre". Cette "**Charte graphique 32 bits** " ne pourra remplacer la précédente qu'à partir du moment où l'armée de terre aura définitivement abandonné Windows 3.11 dans ses préconisations. Ceci n'étant pas encore le cas, cette version n'est, pour le moment, qu'une extension de la précédente, cette dernière restant donc toujours d'actualité.

Ce document est organisé en six parties et comporte:

- **Chapitre 1**: une présentation du processus de conception d'une application graphique interactive,
- **Chapitre 2**: une présentation des spécificités de Windows NT 4,
- **Chapitre 3**: une exposition des principes de base de conception d'une IHM,
- **Chapitre 4**: un descriptif détaillé des différents constituants d'une IHM et quand on doit les utiliser,
- **Chapitre 5**: un guide de conception exposé sous formes de règles,
- **Chapitre 6**: un petit guide de portage des applications de Windows 3.1 vers Windows NT 4 ou Windows 95.

Enfin, une **annexe** est jointe pour permettre, par un jeu de questions - réponses, de mesurer rapidement la qualité de l'interface graphique d'une application développée ou en cours de développement.

Bibliographie et sources

- "Guide de style pour les applications Windows NT 4.0 " de SQL Ingénierie.
- "Les techniques du dialogue ", EDF/GDF.
- "L'interface graphique ", J.M. Gillet, InterEditions.
- "The Model Human Processor ", S. Card, T. Moran, A. Newell.
- "Common User Access, advanced interface design guide ", IBM, 1989.
- "GUI designing & building ", Dr.Dobbs Journal, numéro spécial.

Conception d'une application graphique

Introduction

On ne se lance pas dans la réalisation d'une application graphique comme on pouvait le faire dans le cas d'une application orientée caractères. Il y a désormais deux ***grands*** projets distincts au sein d'une même application en cours de développement: celui lié à la réalisation de l'interface utilisateur d'une part, celui lié aux fonctionnalités de l'application d'autre part. Le processus de développement a donc lui aussi évolué.

Du séquentiel à l'événementiel : une nouvelle façon de développer

Par opposition à une application séquentielle qui dirige l'utilisateur du début à la fin des traitements, une application événementielle est dirigée par l'utilisateur. Cela ne signifie pas pour autant que l'utilisateur puisse faire tout ce qu'il veut. En effet, les différentes actions qu'il peut à son gré, exécuter ou non, sont proposées par le programme. L'application doit donc être en mesure de gérer plusieurs enchaînements différents d'actions.

Deux principes fondamentaux en découlent:

- Les actions possibles à un instant donné doivent être indépendantes, c'est à dire que l'on doit à la limite pouvoir les exécuter toutes en même temps sans rencontrer de problème de concurrence ou de collision.
- La liste des actions possibles ne dépend que du contexte courant dans lequel se trouve l'application à l'instant considéré. Le nombre de contextes différents est donc connu et, bien sûr, les actions hors contexte ne doivent pas être accessibles à l'utilisateur.

Ainsi voit-on réapparaître le concept séquentiel dans une application événementielle. Pour affiner les différences fondamentales entre ces deux types d'application, on retiendra qu'un programme séquentiel suit, dans son déroulement, une succession d'événements, alors qu'un programme événementiel suit une succession de contextes (ou d'états) au sein desquels plusieurs événements indépendants peuvent avoir lieu.

Les conséquences sur le processus de développement sont immédiates. Dans l'application future, le séquencement des contextes doit être décrit par un scénario d'utilisation type. Ce scénario ne peut être décrit et réalisé qu'avec le concours des utilisateurs bénéficiaires de l'application. Le processus de développement doit donc être précédé d'une phase de consultation de ces utilisateurs.

L'infographiste proposera ensuite une maquette. Une maquette est une application passive: seule l'IHM est entièrement codée, les fonctionnalités étant simulées par un codage "en dur". En fait, on peut considérer la maquette comme un tutorial exhaustif de l'application future.

C'est seulement lorsque la maquette sera validée, que le codage proprement dit de l'application pourra commencer. A la livraison du prototype, l'équipe de développement rentrera alors dans un cycle de maintenance (bug + correction = patch) qui ne doit en aucun cas entraîner de modification de l'interface graphique.

Si ce processus de développement n'est pas respecté, il y a de fortes chances pour que le cycle de maintenance devienne le trop classique " essais + erreurs = corrections ". On retiendra donc que dans ce nouveau processus de développement, l'informaticien subit le même sort

que son programme: ce n'est plus lui qui commande, mais bel et bien l'utilisateur; ce que beaucoup ont encore trop tendance à oublier.

Le processus de conception

Première phase: prise en compte de l'activité de l'utilisateur
Cette phase est essentiellement constituée d'un dialogue entre le ou les utilisateurs et le concepteur. Ce dialogue permet d'aboutir à un compromis entre les désirs de l'utilisateur et les possibilités de l'informaticien.

Il comporte les étapes suivantes:
- Définition du scénario et des différents contextes de l'application.
- Définition des écrans principaux correspondant à chacun des contextes. Ne pas confondre écran et fenêtre: un écran peut contenir plusieurs fenêtres...
- Définition des écrans secondaires au sein de chaque contexte (écrans correspondant aux événements majeurs du contexte, ceci ne concerne pas des boîtes de dialogue de demande de confirmation...).

A l'issue des entretiens, le concepteur dispose d'un cahier des charges composé essentiellement de dessins d'écran. Il est alors en mesure de coder la maquette.

Deuxième phase: codage de la maquette
C'est ici qu'intervient l'utilisation d'une charte graphique, garante du respect des standards en matière d'interface homme - machine (standards universels tels CUA89 ou établis en interne). Cette phase peut être extrêmement longue s'il faut coder l'IHM " à la main "... On distingue l'infographiste débutant du confirmé, au nombre total de fenêtres que contient son application. Là où quelqu'un de confirmé n'en utilisera que 50, un débutant en utilisera 150. L'influence sur la taille du code est élevée puisque l'on a pu observer qu'il faut en moyenne 7 lignes de code pour gérer un événement et qu'il peut y avoir beaucoup d'événements par fenêtre.

Parallèlement, les fonctionnalités de l'application peuvent bien sûr déjà être en cours de réalisation par une autre équipe. Mais le projet reste dans cette deuxième phase jusqu'à ce qu'ait été produite une IHM entièrement validée par l'utilisateur client. Il faut d'ailleurs éviter ici les pièges de perfectionnistes et savoir se contenter d'une interface correcte, sans surenchères inutiles et excès d'esthétisme...

Troisième phase: codage du prototype
Il s'agit en fait d'implémenter les fonctionnalités réelles de l'application. A ce stade, l'IHM est entièrement définie et ses modifications ne peuvent être que mineures (changement des call-back passifs en call-back réels). En tout état de cause, cette phase ne doit en aucun cas concerner l'aspect visuel de l'application.

Les spécificités de Windows NT 4.0

Introduction

Les nouveautés en matière d'interface graphique ont en fait été amenées par Windows 95 et portées sur Windows NT 4.0. La nouvelle interface graphique de ces deux systèmes d'exploitation entraîne un double changement d'habitudes, à la fois chez les utilisateurs et chez les programmeurs. Les apports nouveaux de Windows 95 et Windows NT 4.0 ne concernent pas uniquement les composants visuels originaux qui ont pu être introduits, mais également (et pour une part notable) une philosophie nouvelle quant à l'emploi de leur interface graphique. C'est cette nouvelle philosophie et ses conséquences qui sont étudiées ici.

Une nouvelle interface graphique orientée document

La nouveauté de l'interface graphique de Windows NT 4.0 n'est pas seulement le " look and feel " Windows 95, mais surtout la nouvelle philosophie qu'elle amène. En effet, sous Windows 3.x on était habitué au concept d'interface orientée *application*. Sous Windows NT 4.0, on est en présence d'une interface orientée *document*, l'application étant réduite à la notion d'outil (éventuellement parmi plusieurs), utilisé pour traiter le document.

Ce nouveau concept n'est pas sans conséquences sur les IHM utilisées. Il conditionne en particulier les choix entre MDI et SDI qui s'appuieront désormais sur de nouveaux critères.

Concepts d'interface introduits par Windows 3.x

Le modèle SDI

Acronyme de *Single Document Interface*. Avec ce type d'IHM, l'application ne peut ouvrir qu'un seul document. Avant d'en ouvrir un autre, il faut fermer le précédent. Ce type d'interface est arrivé avec la première version de Windows et il existe encore des applications de ce genre livrées en standard (Notepad, MS-Write...). Considérée comme désuète, elle n'est pratiquement plus utilisée aujourd'hui en raison de ses limitations face à des applications de plus en plus performantes.

Le modèle MDI

Acronyme de *Multiple Document Interface*, c'est l'interface de seconde génération. Elle permet d'ouvrir plusieurs fenêtres au sein d'une application. La notion de document peut être élargie selon les fonctionnalités de l'application. Un environnement de développement a tout intérêt à être MDI, donnant ainsi la possibilité d'ouvrir simultanément des fichiers sources ascii, des bibliothèques, des ressources, etc. La fenêtre de l'application devient ainsi le *bureau* de l'utilisateur, offrant la possibilité de stocker temporairement des documents sous forme de fenêtres réduites. En fait le MDI permet simplement de se rapprocher des habitudes de travail manuelles, ce qui explique son succès d'aujourd'hui.

Dans la mesure où il faut d'abord lancer l'application avant de pouvoir ouvrir tous les documents à traiter, l'interface MDI est résolument orientée application et va donc de ce fait à l'encontre des nouveaux concepts de Windows NT 4.

Nouveaux concepts introduits par Windows NT 4.0

Retour du modèle SDI

L'orientation document de Windows NT 4.0 permet de donner un second souffle au modèle SDI. En effet, le simple fait de maintenant cliquer sur le document à traiter amène le système d'exploitation à lancer l'application associée. Il suffit de lancer autant d'instances qu'il y a de documents à traiter (le système étant multi-tâches). L'utilisateur aura alors l'impression d'avoir ouvert plusieurs documents avec la même application, ce qui reprend bien le concept MDI, mais adapté à un système d'exploitation mono-tâche.

Reprise et adaptations du modèle MDI

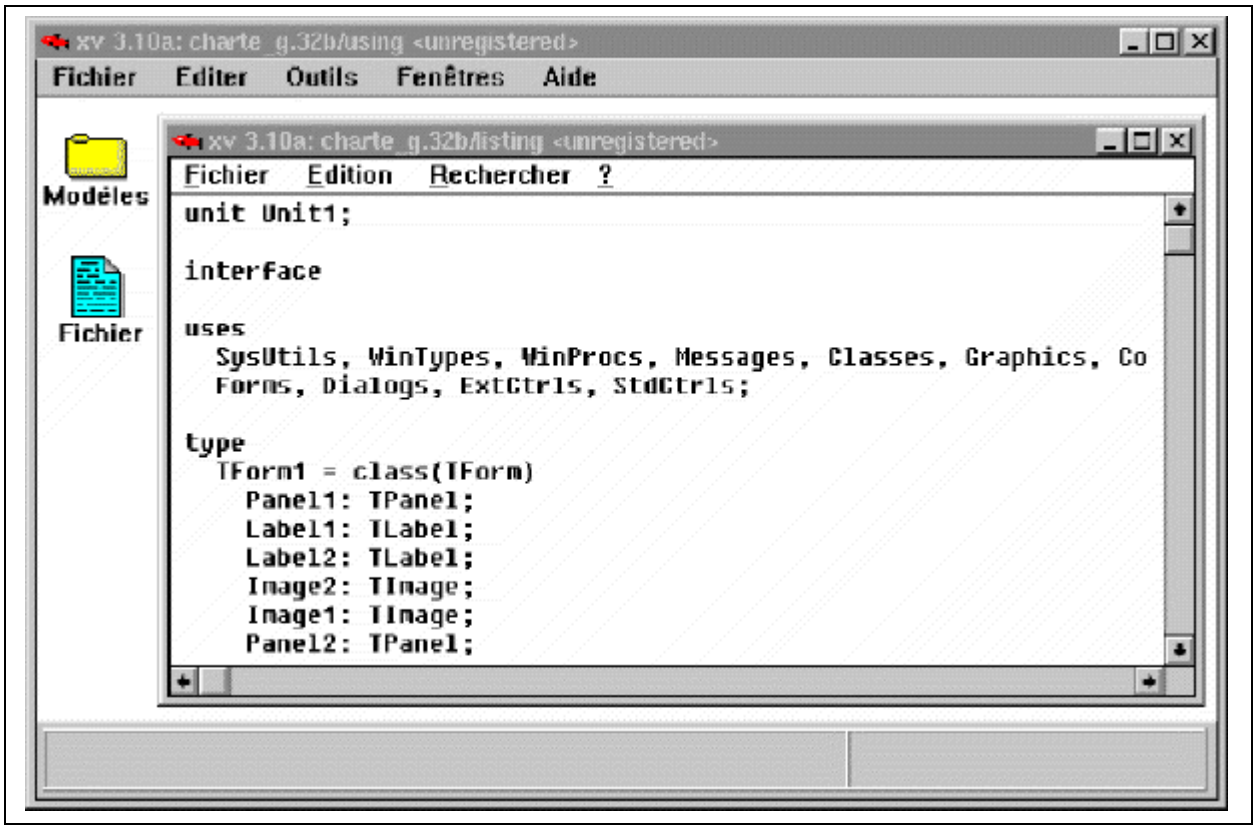
Avec l'arrivée de la nouvelle interface graphique, on pourrait donc penser que le MDI s'efface au profit du SDI. Or, en regardant les nouvelles versions des applications bureautiques, on peut constater qu'il n'en est rien. En fait le modèle MDI restera toujours utilisé pour les applications de gestion alors que le modèle SDI restera réservé aux applications de type outil, et ce pour trois raisons:

1. Contrairement à un outil qui propose plusieurs traitements pour un objet, une application de gestion propose un choix réduit d'actions possibles, mais sur plusieurs objets.
2. Les occurrences d'objets d'un outil sont indépendantes, alors qu'une application de gestion doit gérer les relations entre les objets qu'elle manipule.
3. Un outil propose une vue unitaire d'un domaine fonctionnel, alors qu'une application de gestion en propose une vue globale.

Ainsi, bien qu'en contradiction apparente avec l'esprit de Windows NT 4.0, Microsoft propose toujours le modèle MDI, mais avec des aménagements qui permettent de le rendre plus performant. Ces aménagements sont des nouveaux sous-modèles MDI, à savoir les sous-modèles *Workspace*, *Workbook* et *Project*.

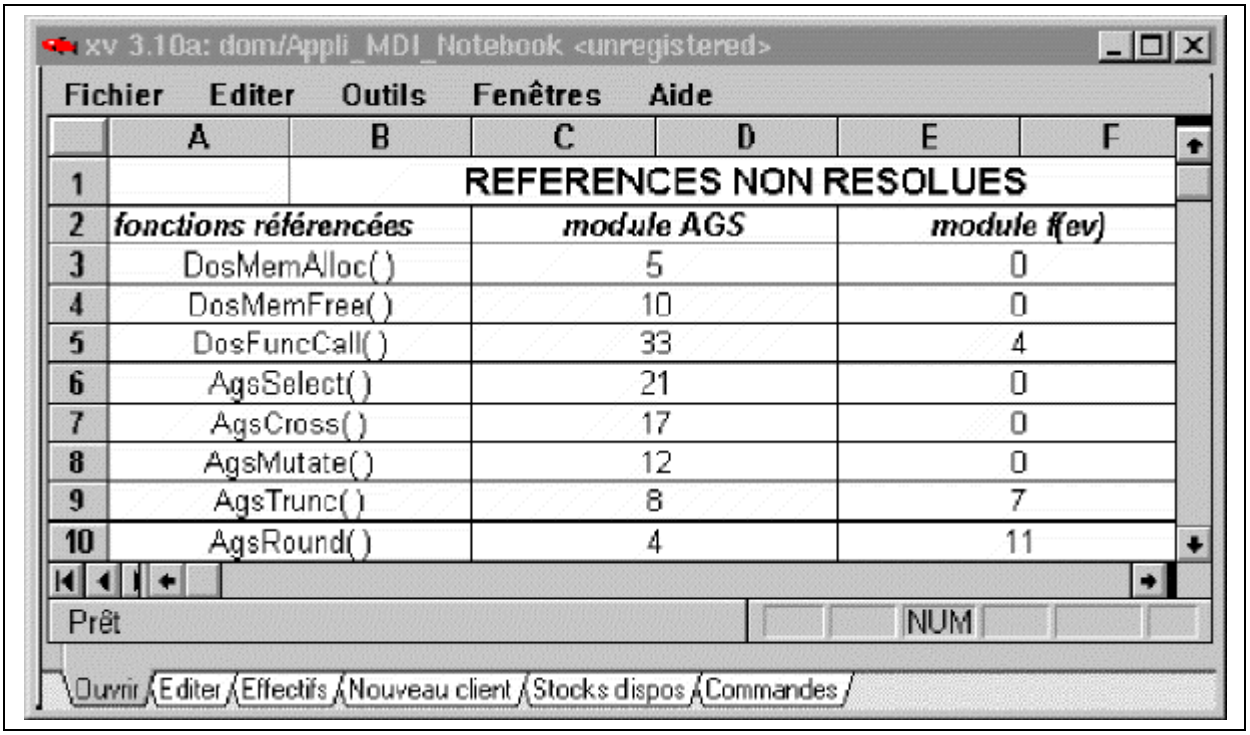
le modèle Workspace

Le modèle s'apparente au MDI classique dont il reprend les caractéristiques originales. Il se veut en fait être un " bureau de travail " pour l'application. En ce sens, ce n'est plus un container de fenêtres seulement, mais de tout type de documents tels que l'entend Windows NT (volumes, répertoires, sous répertoires, fichiers...). Il est possible de sauvegarder l'état de ce " sous-bureau " pour le récupérer tel quel lors du prochain lancement de l'application.



Le modèle Workbook

Le sous-modèle MDI " Workbook " se comporte de façon similaire au modèle Workspace. Il est cependant organisé comme un livre et présente de ce fait ses sous-fenêtres comme les pages de ce livre. Les fenêtres sont toutes ouvertes à leur taille maximale, et leur titre est rappelé sur l'onglet de sélection: cette particularité impose donc une barre de menu et une barre d'outils collectives, identiques à celles que l'on rencontre dans une application MDI classique.



Le modèle Project

Ce modèle s'apparente aux fenêtres dossiers de Windows NT 4.0. Les objets qu'elles contiennent peuvent être ouverts dans des fenêtres filles qui, contrairement à l'esprit MDI classique, ne sont pas liées à l'espace de travail. On retrouve donc chaque fenêtre fille accessible depuis la barre des tâches. Cependant, la fermeture de l'espace de travail (de l'application) entraîne celle de toutes ses fenêtres filles. L'application Microsoft Exchange est un exemple d'application de type " Project ".

Les objets standards d'interface graphique sous Windows NT 4.0

Windows NT a repris les éléments d'interface graphique standards de Windows 3.x (et de Windows NT 3.51). Il a repris également les éléments qui sont devenus des standards de fait et amène un certain nombre de nouveautés. Parmi les standards de fait repris des versions antérieures on trouve:

- barres d'outils avec standardisation des principales icônes,
- barres d'état,
- onglets,
- listes hiérarchiques,
- champs défilant,
- bulles d'aide.

Parmi les nouveautés introduites, on a:

- un objet liste graphique et texte, multi-colonnes avec en-têtes,
- un objet champ de saisie multi-lignes permettant d'intégrer des attributs de mise en forme (police, taille, gras, paragraphes...),
- un objet texte permettant la saisie à partir d'un crayon optique.

En reprenant tout l'existant de l'interface graphique de Windows 3.1x, Windows NT 4.0 permet de maintenir comme étant toujours d'actualité les règles établies pour la production d'IHM. Les infographistes ne devront donc changer leurs habitudes de conception que dans le cas où ils n'auraient pas suivi jusqu'alors les recommandations de la précédente version de la charte graphique...

Règles de base pour construire une IHM

Introduction

Dans ce chapitre, nous allons aborder les points généraux qui régissent la conception d'une interface graphique. Ces principes ne constituent pas un diktat, mais s'appuient effectivement sur des études menées relatives à la personnalité de l'utilisateur.

En effet, lorsque l'on conçoit une IHM, il faut garder à l'esprit que les utilisateurs:

- *utilisent l'application pour effectuer un travail précis, les considérations techniques ne les intéressant pas,*
- *ne voient en fait que ce qui se passe sur leur écran,*
- *aiment et veulent se sentir aux commandes,*
- *n'apprennent que ce qu'ils ont besoin de savoir,*
- *et se souviennent de manière globale*.

Pour définir une interface, il faut donc avant tout comprendre comment raisonne l'utilisateur. C'est pour cette raison que nous étudierons le modèle de l'utilisateur avant d'énoncer les principes de conception qui s'appuient dessus.

Le modèle du processeur humain

Pour comprendre les réactions d'un utilisateur, on se fonde sur une thèse réalisée par trois Américains: *The Model Human Processor*, de S.Card, T.Moran et A.Newel. Ce modèle de processeur humain décrit le fonctionnement du cerveau avec la même approche sémantique que celui des ordinateurs.

Selon ce modèle, l'individu est organisé autour de trois systèmes:

1. ***le système sensoriel,***
2. ***le système moteur,***
3. ***le système cognitif.***

Chacun de ces systèmes dispose d'une mémoire à long terme et d'un mécanisme de commande.

1. **Le système sensoriel** est spécialisé dans le traitement des sens. En ce qui nous concerne, ***la capacité du système visuel est de 17 symboles***, sa persistance est de 200 ms et le temps du cycle du processeur visuel est de 100 ms.

2. **Le système moteur** s'occupe des mouvements, le processeur moteur ayant un temps de cycle de 70 ms. A notre niveau, ce sont les mouvements de la main qui nous intéressent. On retiendra qu'elle peut se déplacer à la vitesse de 1,5 ms^{-1} et que la loi de Fitts nous précise que ***le temps nécessaire pour placer la main sur une cible ne dépend que de la précision requise*** (soit le rapport entre la distance à parcourir et les dimensions de la cible). Ceci est important pour définir la taille des icônes et des boutons.

3. Le système cognitif (ou système mental) a lui aussi un temps de cycle de 70 ms et contrôle le comportement de l'individu. Ce contrôle s'effectue en fonction de ***la mémoire à long terme*** (les connaissances) et de ***la mémoire à court terme*** (les informations). La mémoire à long terme contient la connaissance de l'individu. Sa capacité est infinie et la durée de vie des informations qu'elle contient aussi. Son codage est sémantique: une opération de lecture se fait par reconnaissance ou par association. Pour qu'une lecture puisse avoir lieu dans un minimum de temps, il faut donc présenter un critère de sélection très discriminant. Les informations lues sont alors transmises dans la mémoire à court terme. Cette dernière contient les informations en cours de traitement. Sa capacité est de 7 objets (plus ou moins deux) d'une durée de vie de 7 à 100 secondes. Son codage est visuel ou auditif. C'est en fait un véritable cache entre les divers sous-systèmes.

Le processeur cognitif fonctionne suivant le principe reconnaissance - action. Lors de la phase de reconnaissance, il recherche dans la mémoire à long terme les actions liées au contexte contenu dans la mémoire à court terme, alimentée par le système sensoriel. Pendant la phase d'exécution, il provoque des modifications de celle-ci. Le principal souci d'un infographiste est donc cette fameuse mémoire à court terme, et le modèle du processeur humain permet d'édicter une première règle fondamentale: ***la règle des 7.***

" Le nombre d'objets résidant dans la mémoire à court terme étant limité à sept, on veillera, pour ne pas déclencher d'hésitation chez l'utilisateur, à ne jamais dépasser:

- ***7 menus pour la barre de menus.***
- ***7 items par menus (mettre des traits de séparation fonctionnelle s'il doit y en avoir plus).***
- ***7 lignes simultanées dans une zone de liste fixe.***
- ***7 icônes dans la barre d'outil (avec des séparations physiques s'il doit y en avoir plus). "***

Les principes de base pour créer une interface graphique

Cette partie expose les règles fondamentales qui orientent la conception d'une IHM. Elles ne s'appliquent pas aux objets de l'environnement graphique (ce qui sera traité par le chapitre suivant), mais expliquent les principes fondamentaux à respecter.

Principe de cohérence entre applications

La cohérence consiste à homogénéiser à la fois les apparences et les comportements réactifs entre les applications. Ainsi, face à une nouvelle application, l'utilisateur ne se sent pas perdu et n'a pas à apprendre de nouveaux gestes qui iraient à l'encontre de ses réflexes (mettre l'item "*impression*" dans un menu " *export* " est certes très logique, mais personne ne le fait: il vaut donc mieux ne pas s'aventurer dans cette voie là...). Quatre niveaux de cohérence doivent être observés:

Cohérence avec l'environnement graphique du système d'exploitation

L'environnement graphique de Windows est un environnement *fédérateur* des applications (attention, donc, aux nouveaux arrivants en provenance de l'environnement X...). Il propose toute une collection de contrôles qui lui sont propres et déjà cohérents entre eux: il ne faut utiliser que ceux-ci, à l'exclusion de tout autre (on reparlera de ce détail lors du portage des applications entre Windows 3.1x et Windows 95/NT), en particulier des " petits plus " proposés par les L4G.

Au sein même de Windows, il y a de nombreux exemples qui indiquent la marche à suivre en matière d'ergonomie.

Cohérence avec les autres applications

Toutes les applications de Windows sont construites à partir de composants élémentaires (fenêtres, boîtes de dialogue, menus, contrôles,...) qui sont somme toute en nombre assez réduit. Les applications devraient donc toutes se ressembler. Hélas, c'est loin d'être le cas, car ces composants de base sont extraordinairement paramétrables. Ce sont autant de libertés laissées aux concepteurs débutants qui ne manqueraient pas d'en abuser pour aboutir à des IHM sans le moindre point commun avec l'application " de référence ". Il suffit de regarder les fenêtres principales de " MS-Powerpoint " et " MS-Word " pour se rendre compte jusqu'où il est possible de pousser la cohérence entre applications pourtant fort différentes.

Cohérence à l'intérieur même de l'application

En interne, la cohérence s'exprime par l'uniformité des présentations et des comportements: mêmes polices pour mêmes fonctions, mêmes décorations des fenêtres filles, etc. Il n'y a rien de plus rebutant que de voir un bouton *OK* tantôt à gauche du bouton *Annuler*, tantôt à droite, selon la boîte de dialogue qui l'affiche.

Cohérence avec le monde réel

Une application a pour but premier d'informatiser un métier qui existait souvent bien avant l'ordinateur. Elle doit donc être le reflet du monde réel qu'elle synthétise. L'utilisation de métaphores est plus aisée pour l'utilisateur que des panneaux indicateurs

(un agenda pour gérer des rendez-vous, un stylo ou un pinceau pour dessiner... sont des garanties d'une appropriation aisée de l'application par les utilisateurs).

Principe de retour d'information

Le retour d'information ou *feed back* en anglais permet à l'utilisateur de recevoir une sorte d'accusé de réception après avoir déclenché une action. Ce retour d'information ***est obligatoire et immédiat*** pour ne pas laisser l'utilisateur dans l'expectative. Il peut être bref ou non, de type métaphore ou message, selon la nature de l'action engagée.

Il existe encore beaucoup trop d'applications sous Windows 3.1x où le seul moyen d'obtenir ce retour d'information après avoir lancé une action est d'observer l'activité du disque dur à travers la LED en face avant du micro...

Faire patienter ou non l'utilisateur

La durée du feed-back doit être égale à la durée des traitements. Le feed-back est instantané pour les actions très courtes (enfoncement d'un bouton par exemple). Il prend la forme d'un sablier pour des traitements inférieurs à 5 secondes ou la forme d'une barre de progression pour les traitements plus longs.

Les messages de Windows NT 4.0

Destinés aux utilisateurs, ils doivent être clairs et non répressifs. Windows NT 4.0 connaît quatre types de messages, différenciés par des graphismes spécifiques, tous modaux.

- Le message d'information (" **i** "). Il est utilisé pour donner un compte-rendu immédiat sur le résultat d'une action importante ou qui a duré un certain moment.

- Le message d'indication de progression, qui informe l'utilisateur de l'état d'avancement des travaux en cours.

- Le message d'avertissement (" **!** "). Il prévient l'utilisateur qu'une décision doit être prise avant de lancer une action déterminée. Les demandes de confirmation d'actions irréversibles entrent dans cette catégorie.

- Le message d'erreur bloquante (" **x** ") précise qu'une anomalie a été détectée et qu'elle doit impérativement être corrigée avant de pouvoir poursuivre.

Ces deux derniers messages sont à utiliser avec parcimonie. Selon le vieil adage " mieux vaut prévenir que guérir ", l'application ne devrait pas permettre que des situations critiques puissent survenir. Ils doivent de plus être positifs et proposer des solutions (même sous-entendues) plutôt que de constater des échecs.

Exemple:
- correct : **" enregistrement impossible: disque plein "**
- incorrect : **" erreur de copie sur disque "**

Principe de sobriété

Devant la multitude de possibilités qu'offre un environnement graphique pour la présentation des IHM, le développeur débutant est souvent pris de vertiges et fini par trop en faire. Ce phénomène est connu sous le nom de " S*yndrome du sapin de Noël "* et engendre des applications aux polices illisibles, aux couleurs trop vives, aux contrastes inappropriés, à des fenêtres particulièrement surchargées: en un mot à des IHM certes jolies, mais inexploitables. Il y a également dans ce domaine quelques principes à respecter.

La couleur

Elle n'est pas neutre et apporte plus d'inconvénients que d'avantages: on essaiera de l'éviter dans l'IHM. Dans le cas général, on notera sa règle d'emploi: *" la couleur doit être un moyen redondant avec un autre mode de codage pour l'apport d'information, et non constituer l'unique moyen de discrimination de cette information ".*

De plus, il faudra respecter l'usage du codage " culturel " de la communauté humaine:

- rouge = alerte,
- jaune = attention,
- vert = aucun danger,
- blanc = neutre, aucun apport d'information particulier.

Là aussi on pourra se référer aux exemples précédents (*Word* et *Powerpoint*) pour y voir des IHM réussies, bien que réalisées uniquement en nuances de gris. On remarquera que les faces avant d'instrumentations (tableau de bord de véhicules, façades de matériels HI-FI, etc.) respectent depuis longtemps ce principe de sobriété...

Les effets 3D

Contrairement à ce que l'on pourrait être tenté de faire, l'utilisation des effets 3D ne concerne *en aucun cas* l'agrément de l'IHM. Ces effets ne doivent être utilisés que dans le contexte du retour d'information. C'est ainsi qu'on les retrouvera dans les boutons qui s'enfoncent lorsque l'on "appuie" dessus, accentuant l'effet de sensation pour l'utilisateur.

Les textes (la notation hongroise) et les polices

La " notation hongroise " est une convention respectée par les programmeurs C. Elle a été créée par C. Simonyi, de Microsoft, qui est d'origine hongroise. Elle est utilisée pour définir des noms de variables, de fenêtres,... pour tout ce qui doit être baptisé en rappelant sa fonction.

Elle se résume à 4 règles:

- ***la première lettre d'un mot est en majuscule,***
- ***les suivantes sont en minuscules,***
- ***le mot est écrit en entier,***
- ***il n'y a aucun séparatif, ni particule.***

Ainsi, une colonne "client société" dans une table sera présentée ClientSociété, une colonne "nom du client", NomClient, etc. En revanche, les textes informatifs doivent être écrits en langage naturel. Il vaut mieux ne rien écrire plutôt qu'afficher un message comme: "err. n° 23". Dans tous les cas, il faut proscrire les abréviations ou acronymes de toutes sortes, sauf s'ils font partie du vocabulaire courant du métier informatisé (TVA dans une application de comptabilité ne choquera personne...).

En ce qui concerne les polices à utiliser dans les objets constituant l'IHM, il n'en est qu'une seule de permise: la police du système d'exploitation. Elle sera toujours disponible sur toutes les plates-formes sur lesquelles l'application devra être déployée. Etant celle employée par toutes les applications, on respecte ainsi le second principe de cohérence.

Principe de droit à l'erreur

" Si l'utilisateur a fait une erreur, c'est la faute du développeur: l'IHM n'aurait jamais dû le lui permettre ". Cette affirmation est certes un peu exagérée, mais elle doit conduire les développeurs à faire preuve d'humilité. L'interface présente des défauts, l'utilisateur aussi: tout le monde peut se tromper. L'application doit donc être indulgente et non répressive et permettre un certain niveau de retour en arrière. Dans tous les cas elle doit *contrôler* le bien fondé de certaines actions critiques de l'utilisateur en demandant leur confirmation.

Principe de modalité

La modalité décrit un mode de fonctionnement particulier de l'interface. Le confort d'utilisation d'une application dépendra pour beaucoup de l'art avec lequel le développeur aura abordé ce concept.

Les situations modales

La définition en est simple: une situation est modale si l'utilisateur ***doit impérativement*** faire une action avant de pouvoir poursuivre. On la retrouve lorsque l'utilisateur doit introduire des données ou doit accuser réception d'un message. Dans une situation modale, il perd donc toute liberté d'action.

Savoir éviter les situations modales

Les situations modales vont à l'encontre du concept événementiel des interfaces graphiques en imposant à l'utilisateur des points de passage obligés. Elles sont donc à éviter dans la mesure du possible. La plupart des actions modales arrivent via des boîtes de dialogue pour demander des données. Il est utile de se poser la question de savoir si un " tableau de bord " avec des valeurs par défaut affichées en permanence et modifiables à n'importe quel moment ne serait pas plus judicieux.

Toute application de dessin doit proposer une palette de couleurs que l'utilisateur peut choisir à volonté, plutôt qu'un bouton " couleur " qui entraîne l'apparition d'une boîte de dialogue pour l'introduction de celle qu'il souhaite sélectionner...

Contextes inévitables

Il n'y a que trois cas où une situation modale est inévitable, voire obligatoire:

1. Les **dialogues de continuation**. Ils sont employés quand l'application a besoin d'une information indispensable et *locale au contexte* (qui ne peut être prédéfinie par défaut): un nom de fichier ou le nombre de copies à imprimer.
2. Les **messages** envoyés à l'utilisateur qui doivent tous être suivis d'un accusé de réception.
3. Les **dialogues transactionnels**. C'est le cas le plus courant en informatique de gestion. En effet, si les paramètres introduits par l'utilisateur entraînent des modifications d'une base de données, l'application va devoir poser les verrous appropriés sur l'enregistrement concerné. Avec une situation modale, l'utilisateur est obligé de répondre immédiatement, ce qui réduit d'autant la durée de monopolisation de la table en cours de modifications. ***Ce serait une erreur grave que de laisser l'utilisateur dans une situation non modale dans ce contexte particulier***.

Principe du dévoilement progressif

On a déjà sous-entendu ce principe lorsqu'on a donné la définition d'une application événementielle: c'est une séquence de contextes au sein desquels plusieurs actions sont possibles à la discrétion de l'utilisateur. Le principe du dévoilement progressif repose sur une règle assez simple:

" Tant que l'on n'est pas entré dans un certain contexte, il est inutile de présenter les actions qui s'y rapportent. "

Le fait de présenter des actions impossibles à réaliser entraîne inévitablement un sentiment de frustration chez l'utilisateur. Devant la complexité toujours croissante des applications, on en est même arrivé ***à proscrire les composants graphiques grisés*** (les objets momentanément indisponibles). En effet, puisqu'il faut maintenant expliquer en permanence ce qui se passe, si l'on tient aux items grisés, il faut leur prévoir une aide contextuelle justifiant pourquoi ils sont (pour le moment) inutilisables. C'est une augmentation de complexité pas toujours justifiée et c'est pourquoi il vaut mieux ne pas les faire apparaître du tout.

Les widgets de Windows NT4.0

Introduction

Avant d'exposer en détail les règles qui régissent l'emploi des différentes widgets de Windows NT 4.0 / 95, il est nécessaire de bien les connaître, c'est à dire savoir les situer dans l'environnement graphique, connaître leur rôle, leurs caractéristiques et leur mode d'emploi.

Ce chapitre présente l'ensemble des composants visuels par classe d'objet :

- **la classe des fenêtres,**
- **la classe des contrôles,**
- **la classe des menus.**

Cette présentation se fait sous trois angles :

1. ***les rôles,***
2. ***les caractéristiques,***
3. ***les propriétés relevant de la responsabilité du programmeur.***

La classe fenêtre

La classe fenêtre regroupe tous les constituants d'une fenêtre affichée, de la barre de titre aux dessins des bords de celle-ci.

Primary window SDI

Rôle :

- Contient les informations traitées par l'application.

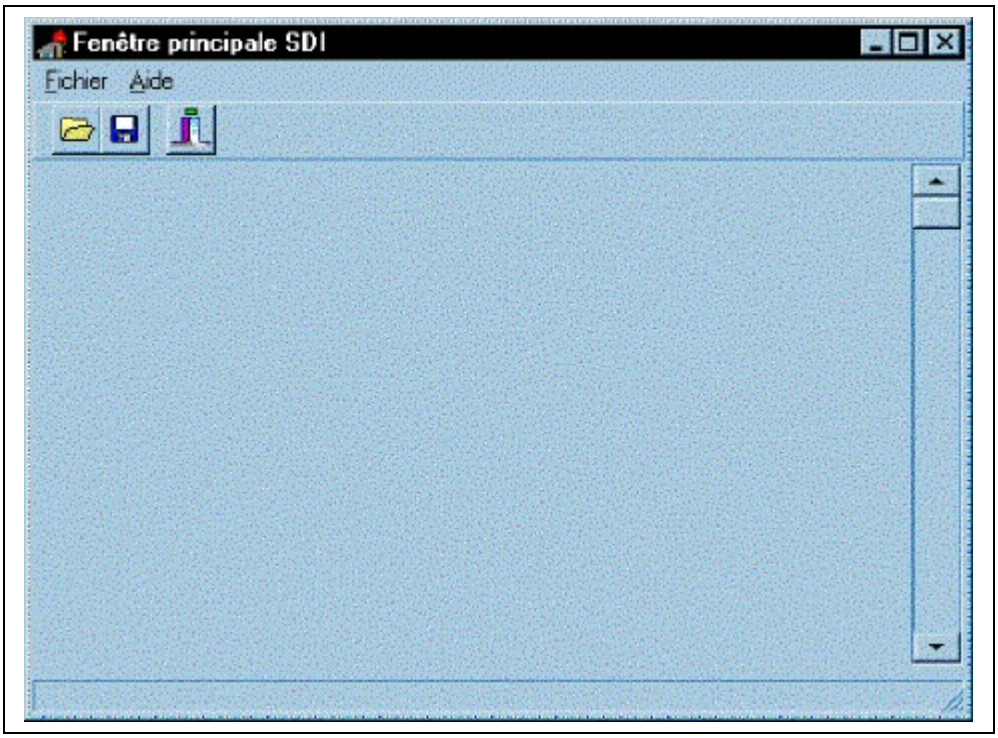
Caractéristiques :

- déplaçable et redimensionnable,
- barre de titre,
- icône de l'application,
- boutons Fermeture, Réduction, Agrandissement et Restauration,
- non modale.

Programmation :

- Définir le titre, l'icône, la barre de menu,
- Eventuellement une barre d'état et une barre d'outil.

Exemple :



Primary parent windows MDI

Rôle :

- Délimite un espace de travail. L'application peut charger plusieurs documents, ou plusieurs instances d'un même document.

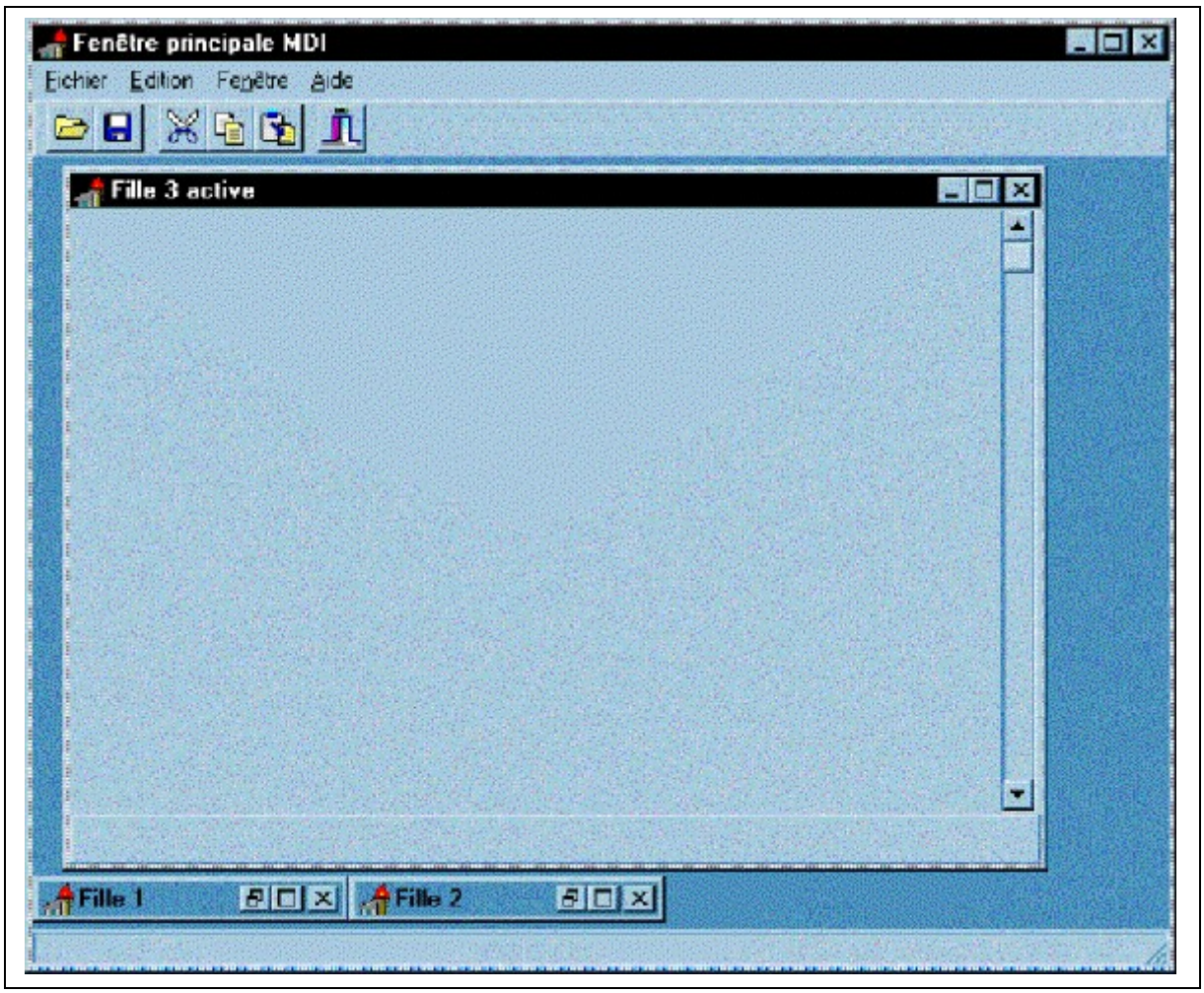
Caractéristiques :

- déplaçable et redimensionnable,
- barre de titre,
- icône de l'application,
- boutons Fermeture, Réduction, Agrandissement et Restauration,
- non modale.

Programmation :

- Définir le titre, l'icône, la barre de menu,
- éventuellement une barre d'état et une barre d'outils.

Exemple :



Primary window/Document window/Child window

Rôle :

- Représente le contenu d'un document (ou d'une de ses instances), de n'importe quelle nature.

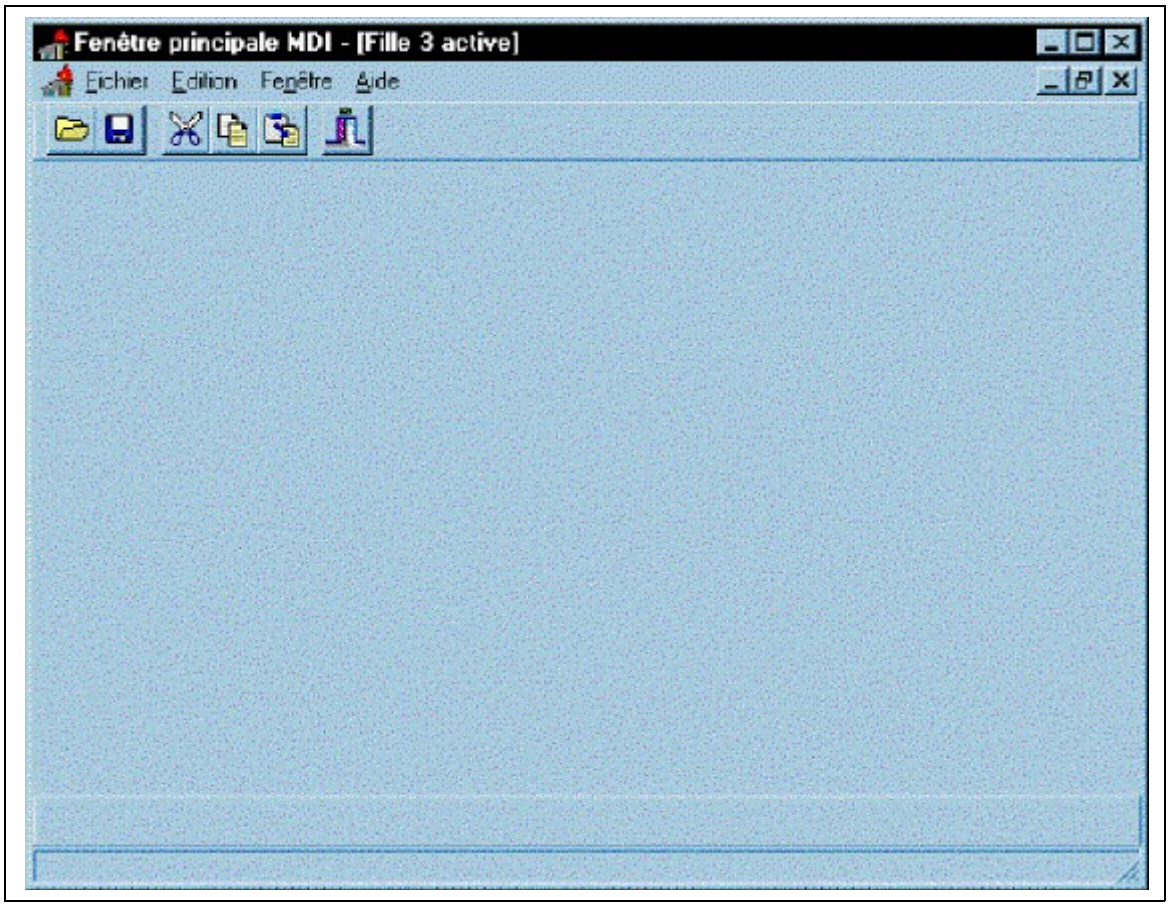
Caractéristiques :

- déplaçable et redimensionnable,
- barre de titre,
- icône de l'application,
- boutons : Fermeture, Réduction, Agrandissement et Restauration, non modale,
- liée à la fenêtre MDI principale incluse dans son espace de travail,
- peut occuper tout l'espace de travail de la fenêtre mère MDI en activant le bouton Agrandissement. Dans ce cas, le titre de la fenêtre fille est combiné à celui de la fenêtre principale. Le menu et la barre d'outils disponibles sont alors ceux de la fenêtre fille.

Exemples :

- L'exemple précédent montre le cas général.

- L'exemple ci-dessous montre le cas d'une fenêtre fille agrandie.



Secondary window/Dialog Box

Rôle :

- Permet l'échange d'informations entre l'utilisateur et l'application.

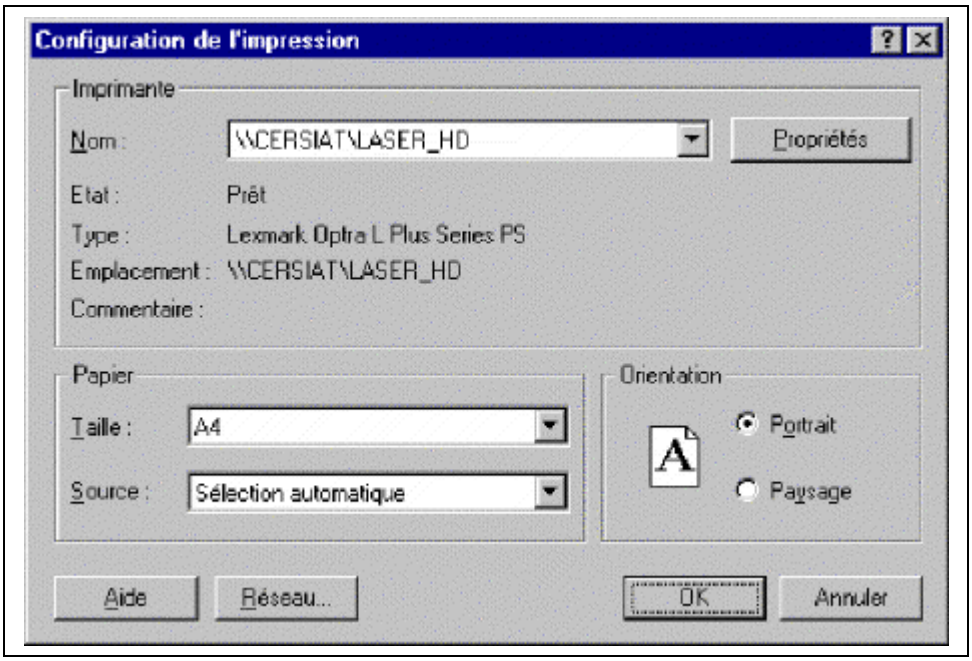
Caractéristiques :

- déplaçable, et non redimensionnable,
- barre de titre,
- bouton de Fermeture (et éventuellement le bouton Info " ? "),
- modale dans la plupart des cas.

Programmation :

- Définir le contenu et les réactions.

Exemple :



Palette window

Rôle :

- Permet de présenter une palette d'options, style tableau de bord.

Caractéristiques :

- déplaçable,
- barre de titre,
- bouton de Fermeture,
- non modale,
- toujours affichée au premier plan.

Programmation :

- Définir le contenu.

Exemple :

Popup window

Rôle :

- Afficher des informations complémentaires contextuelles, telles que les bulles d'aide.

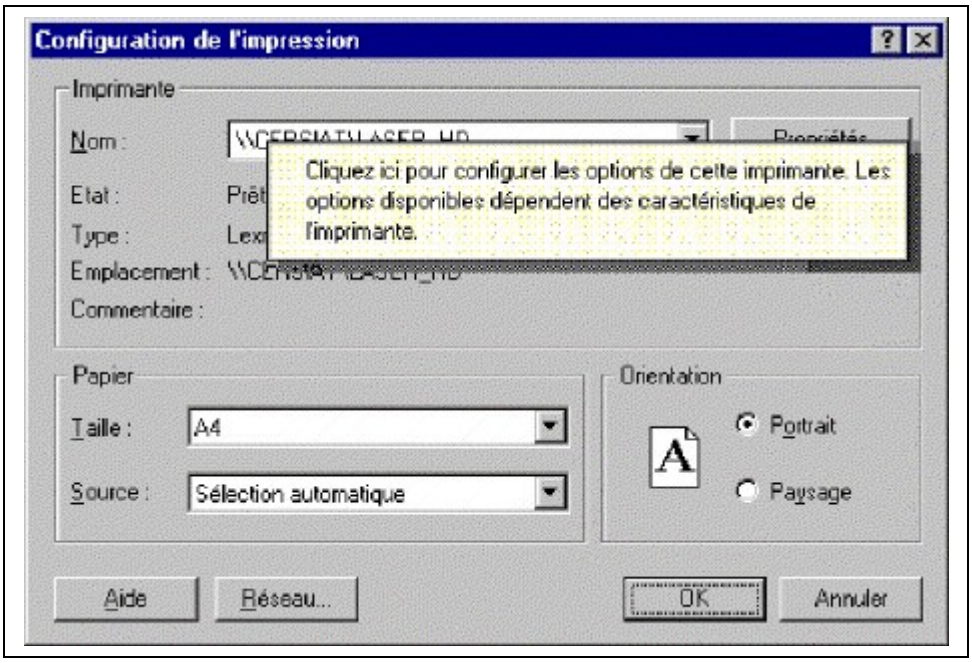
Caractéristiques :

- affichée en fonction de la position de la souris,
- non déplaçable,
- pas de barre de titre,
- aucun bouton (système ou autre),
- modale,
- disparaît dès que l'utilisateur opère une action.

Programmation :

- Définir le texte à afficher.

Exemple :



Attributs des fenêtres

Ces attributs font partie des objets de la classe fenêtre. Toutefois leur comportement est entièrement géré par le système.

On dispose de quatre attributs:

- ***Title bar*** *:* La barre de titre,
- ***Application icon*** *:* L'icône associée à l'application,
- ***Windows Border*** *:* La frontière de la fenêtre (pour redimensionnement...),
- ***System buttons*** *:* Les boutons d'iconisation, de fermeture, d'agrandissement.

Seules les barres de titre et les icônes d'application nécessitent un travail de programmation.

La classe contrôle

Cette classe comprend un très grand nombre de contrôles: tous les objets composant les fenêtres et les dialogues en font partie. Seuls les contrôles standards sont proposés, les autres (les *Custom Controls*), non standards par définition, sont ignorés.

Command button

Rôle :

- Déclenche une action de la part de l'utilisateur.

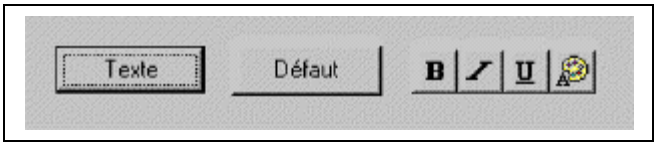
Caractéristiques :

- Lorsqu'on appuie sur le bouton, il s'enfonce (aspect graphique). C'est seulement quand on le relâche que le call-back associé est exécuté.

Programmation :

- Call-back et aspects du bouton (normal, enfoncé, indisponible).

Exemple :



Option button

Rôle :

- Permettent de saisir et / ou visualiser des choix multiples exclusifs.
- Leur utilisation permet dans beaucoup de cas de s'affranchir de boîtes de dialogue modales si des définitions par défaut sont possibles.

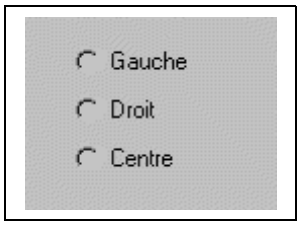
Caractéristiques :

- C'est un bouton-radio, donc un flag binaire indiquant s'il est coché.

Programmation :

- Entièrement du ressort du programmeur. NB: si un choix n'est pas possible dans le contexte considéré, *il vaut mieux ne pas faire apparaître le bouton plutôt que de le griser*.

Exemple :



Check box

Rôle :

- Saisir et / ou visualiser des choix multiples non exclusifs.

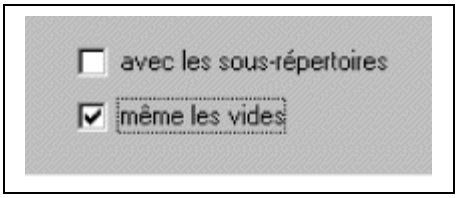
Caractéristiques :

- Comme précédemment sauf que le fait de sélectionner une case n'a pas d'effet (de désélection) sur les autres.

Programmation :

- Ici aussi, tout son comportement est du ressort du programmeur.

Exemple :



Single selection List box

Rôle :

- Permet une sélection simple dans une liste d'éléments. Ce contrôle est utilisé pour la saisie guidée.

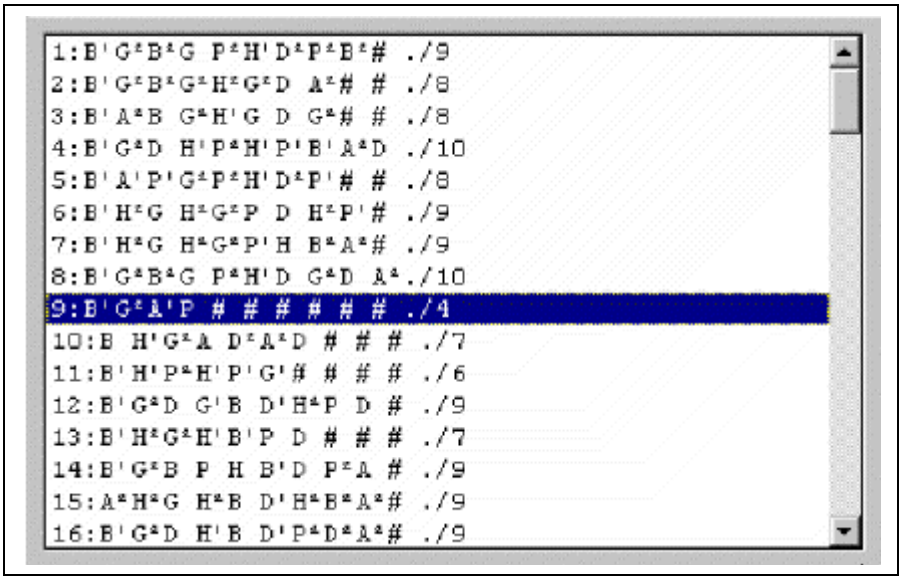
Caractéristiques :

- La sélection est matérialisée par une vidéo inverse. Pour les longues listes, l'ascenseur apparaît automatiquement.

Programmation :

- Contenu de la liste uniquement.

Exemple :



Extended and multiple selection List box

Ce contrôle, comme son nom l'indique, est une extension du précédent. C'est à dire qu'au lieu d'avoir un seul choix possible dans la liste, il est possible de sélectionner plusieurs rangées consécutives. Le reste des caractéristiques est identique à ce qui précède.

Drop down List box

Rôle :

- Permet une sélection simple dans une liste d'élément.

Caractéristiques :

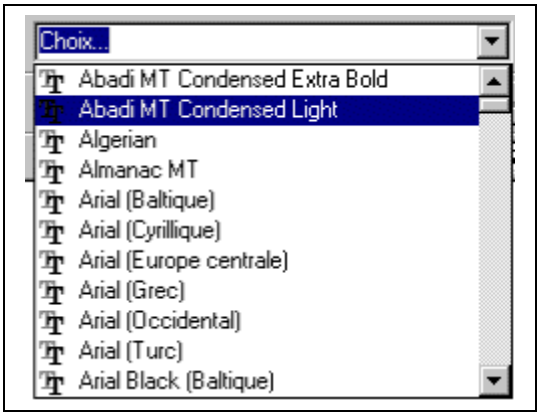
- Similaire à la boîte de liste, mais le contenu n'apparaît qu'en appuyant sur le bouton de défilement.

Programmation :

- Remplir la liste et les valeurs par défaut.

Exemple :

(ici, la boîte de liste est présentée ouverte)



List box view

Rôle :

- Permet une sélection simple dans une liste d'éléments caractérisés par une image et un texte. Une " list box " dispose de trois vues:
  1. *vue par icônes standards,*
  2. *vue par listes,*
  3. *vue par icônes réduites et textes.*

Caractéristiques :

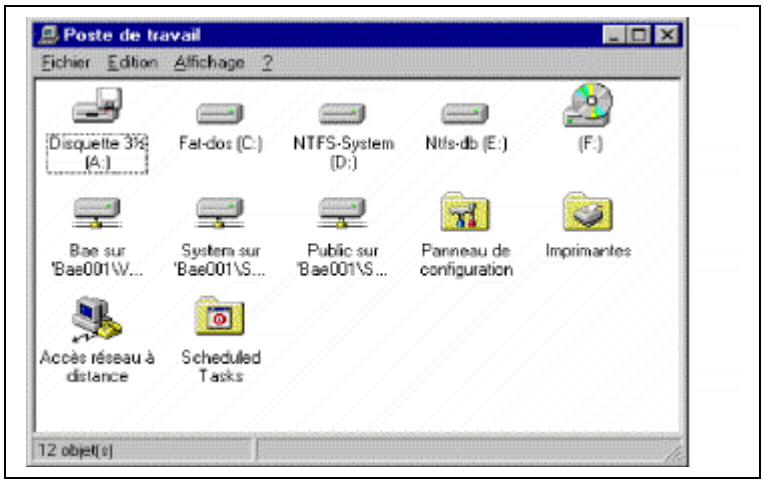
- Suite à une sélection, le libellé apparaît en vidéo inverse.

Programmation :

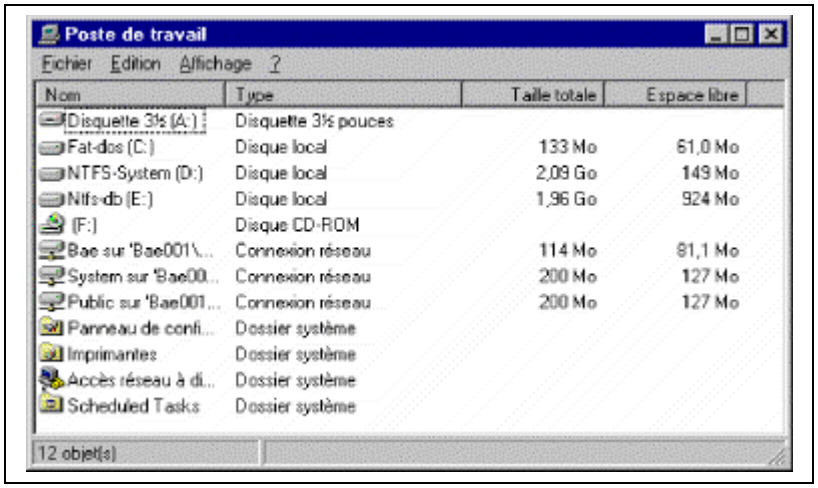
- Contenu de la liste à gérer.

Exemples :

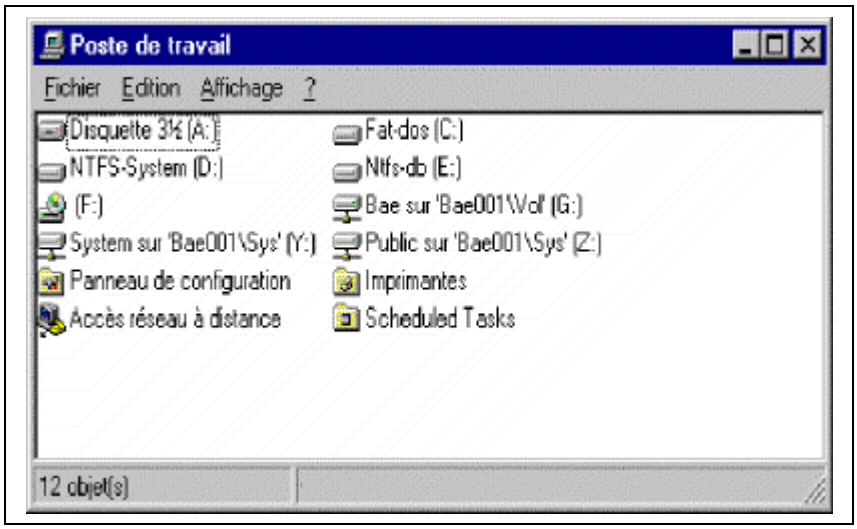
1. vue par icônes standard



2. vue par liste détaillée



3. vue par icônes réduites et textes



List box tree view

Rôle :

- Permet la visualisation et la navigation au sein d'une arborescence.

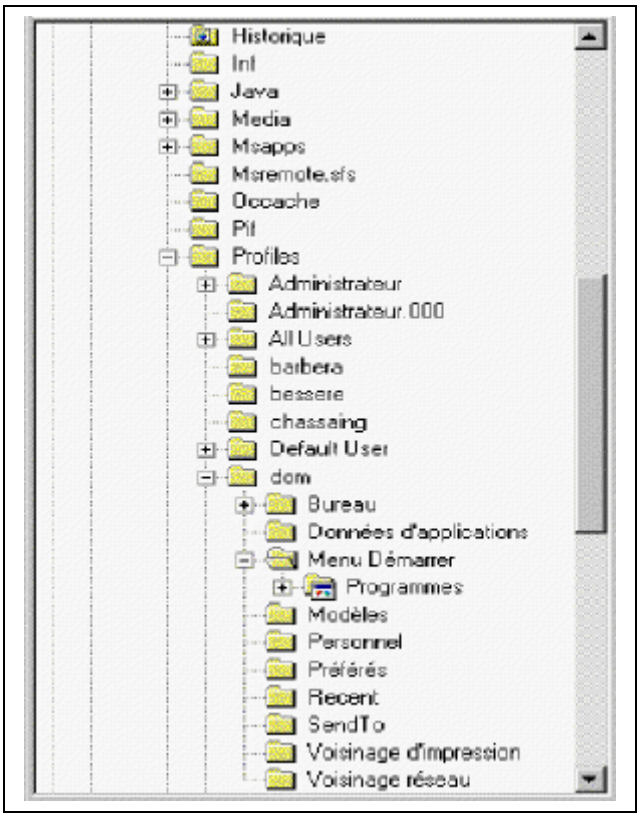
Caractéristiques :

- A l'initialisation, la liste contient tous les éléments de premier niveau.
- Le fait d'appuyer sur " + " ouvre l'arborescence sélectionnée, le fait d'appuyer sur " - " ferme l'arborescence sélectionnée.

Programmation :

- Contenu de la liste à renseigner.

Exemple :



Text box

Rôle :

- Permet la saisie de texte sur une seule ligne.

Caractéristiques :

- Le curseur devient celui du mode texte. Le scrolling horizontal peut être automatique si les dimensions de la boîte sont trop faibles; mais ceci *ne doit jamais se produire*.

Programmation :

- Définir le type de la saisie et éventuellement le nombre maximal de caractères pouvant être saisi.

Exemple :

Rich text box

Rôle :

- Comme précédemment, mais sur plusieurs lignes.

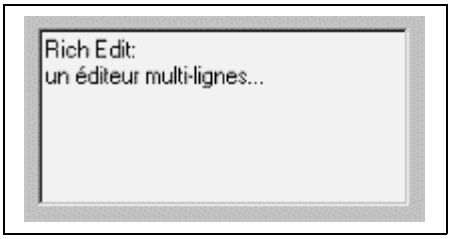
Caractéristiques :

- C'est en fait un véritable éditeur de textes, avec entre autres des possibilités de changement de police.

Programmation :

- Néant.

Exemple :



Combo box

Rôle :

- Permet la sélection simple dans une liste d'éléments ou saisie directe.

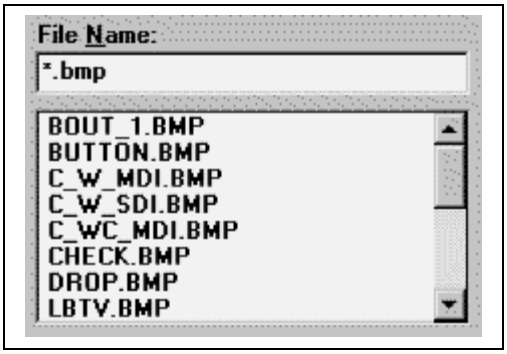
Caractéristiques :

- Comme une " list box " pour la partie inférieure, comme un champ de saisie pour la partie supérieure.

Programmation :

- Remplir la liste.

Exemple :



Drop down Combo box

Rôle :

- Permet la sélection simple dans une liste d'éléments ou la saisie directe.

Caractéristiques :

- Similaire à la " combo-box ", mais le contenu de la liste n'est affiché que si on clique sur le bouton de défilement vertical.

Programmation :

- Remplir la liste.

Exemple :

Spin box

Rôle :

- Permet la saisie directe d'une valeur ou défilement par une " molette ".

Caractéristiques :

- L'action sur les flèches incrémente ou décrémente le compteur.

Programmation :

- Définir la valeur initiale, les pas d'incrémentation - décrémentation, et les bornes inférieure et supérieure.

Exemple :

Up-down

Rôle :

- Permet de modifier une valeur sans saisie, par " molette " uniquement.

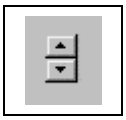
Caractéristiques :

- Ne peut en fait être utilisé seul...

Programmation :

- Définir le pas d'incrémentation - décrémentation de la valeur du champ associé à ce contrôle.

Exemple :



Static text field

Rôle :

- Apporte une information, légende,... à caractère permanent.

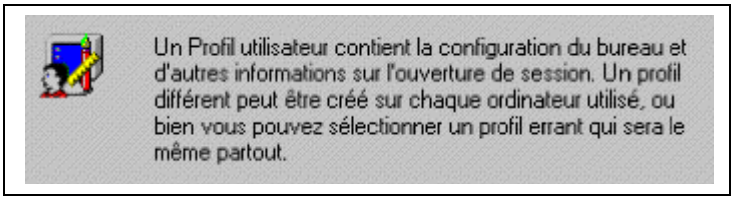
Caractéristiques :

- Le texte est toujours justifié à gauche.

Programmation :

- Aucune, il n'y a aucun lien avec l'application.

Exemple :

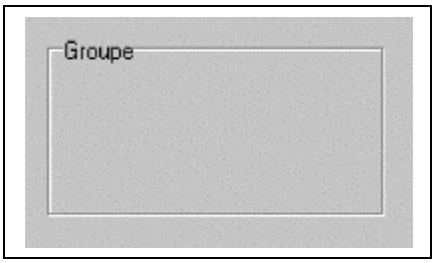


Group box

Rôle :

- Permet de regrouper logiquement un ensemble de contrôles.

Exemple :



Column heading

Rôle :

- Fournit des informations sur le contenu d'une liste de choix visuel affichée sous forme détaillée.

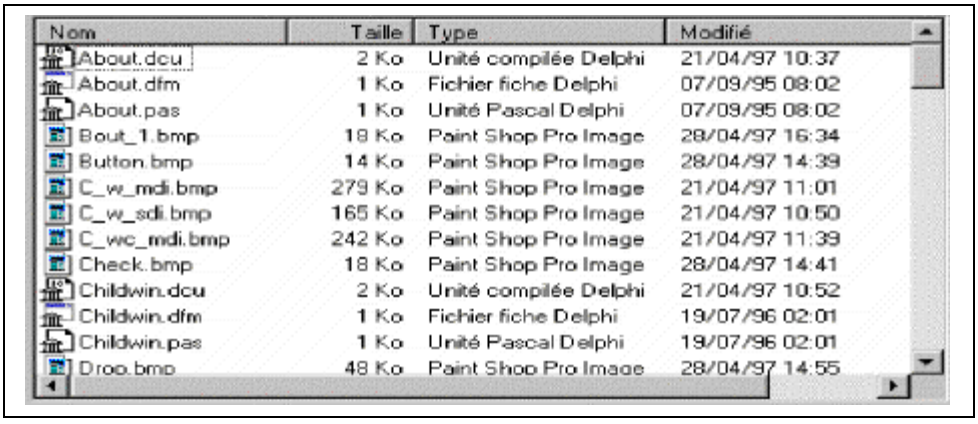
Caractéristiques :

- Les titres peuvent aussi être des boutons.

Programmation :

- Programmation des call-backs si on a opté pour les boutons.

Exemple:



Tabs

Rôle :

- Permet de répartir des affichages sur plusieurs pages au sein d'une même fenêtre.

Caractéristiques :

- Un clic sur l'onglet affiche les informations qui y sont associées.

Programmation :

- Définir le nombre d'index et leur titre.

Exemple :

Slider/Trackbar

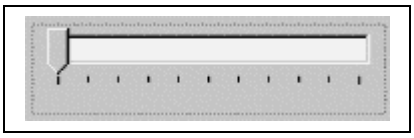
Rôle :

- Permet le réglage grossier de la valeur d'un paramètre.

Programmation :

- Définition de l'orientation horizontale ou verticale, des dimensions de l'indicateur et les bornes inférieure et supérieure des valeurs qui peuvent être choisies.

Exemple :



Progress Indicator

Rôle :

- Affiche par une animation l'état d'avancement d'une action.

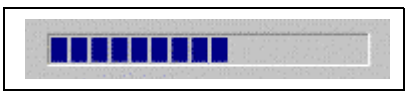
Caractéristiques :

- Affichage sous forme de jauge horizontale. Son emploi est ***obligatoire*** pour les traitements d'une durée supérieure à 5 secondes.

Programmation :

- C'est le développeur qui doit gérer l'avancement de l'indicateur.

Exemple :



Well

Rôle :

- Permet la visualisation de choix exclusifs les uns des autres.

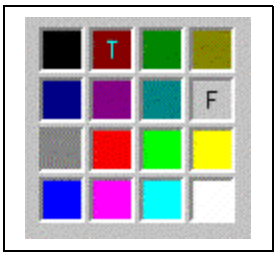
Caractéristiques :

- Fonctionnement identique à celui des boutons radio. Plusieurs indicateurs indépendants peuvent être simultanément utilisés.

Programmation :

- Comme pour les boutons radio.

Exemple :



Toolbar

Rôle :

- Fournit des raccourcis visuels aux actions réalisables depuis les menus.

Caractéristiques :

- Elles sont fonction des contrôles qui équipent la " tool bar ".

Programmation :

- Définition du contenu de la barre d'outils et des call-back associés.

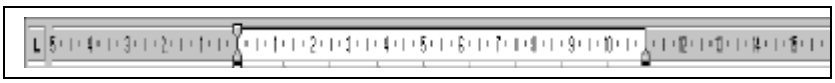
Exemples :

3 grands classiques

1. Ruban

2. Barre d'outils

3. Règle



Status bar

Rôle :

- Donne des informations contextuelles sur l'état d'une application.

Programmation :

- Gérer les informations contextuellement.

Exemple :

Tooltip

Rôle :

- Affiche un texte dans une petite popup pour apporter des informations plus précises sur une image.

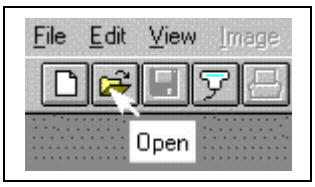
Caractéristiques :

- La bulle apparaît quand le pointeur arrive sur l'image concernée.

Programmation :

- Définir le texte à afficher.

Exemple :



Autres contrôles

Il existe également des contrôles standards pour la gestion d'un crayon optique. Ces derniers (saisie de texte et saisie de dessin à partir d'un crayon optique) ne sont pas traités ici en raison de leurs caractères très spécifiques.

La classe menu

Cette classe regroupe tous les composants constituant les différents types de menus que l'on peut trouver au sein d'une application. Chaque item de menu ou de sous-menu déclenche obligatoirement une action. Il est donc évident que dans le travail de programmation, il y aura les call-backs associés à définir: on ne le répétera pas à chaque fois dans les tâches de programmation à réaliser.

Menu bar

Rôle :

- La barre supporte les différents menus de l'application.

Caractéristiques :

- Elle occupe toute la largeur de la fenêtre et est positionnée sous la barre de titre.

Programmation :

- Définir son contenu. Les réorganisations selon la taille de la fenêtre de l'application sont entièrement prises en compte par le système.

Exemple :

Menu Title

Rôle :

- Informer sur le contenu du menu.

Caractéristiques :

- C'est le point d'entrée du menu. Le menu s'ouvre quand on clique sur son titre.

Programmation :

- Aucune, entièrement géré par le système.

Exemple :

Menu item

Rôle :

- Informer sur l'action qui lui est associée.

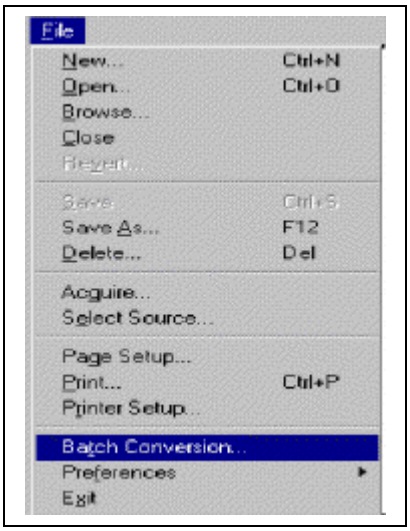
Caractéristiques :

- C'est lui qui déclenche l'action.

Programmation :

- Définir le ***raccourci clavier obligatoire***.

Exemple :



Drop down menu

Rôle :

- Offrir sous forme d'items un ensemble d'actions réalisables.

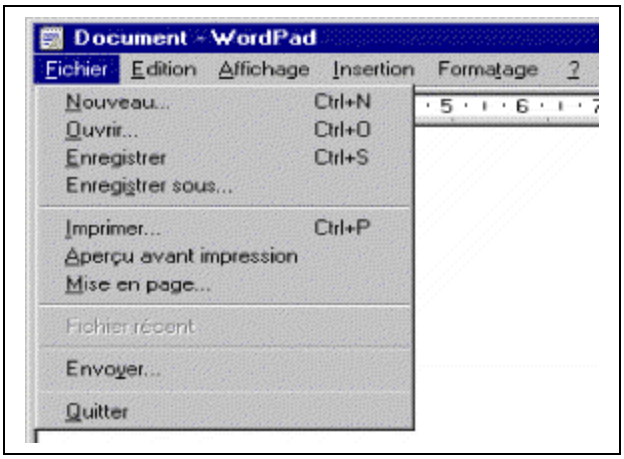
Caractéristiques :

- Le menu déroulement s'ouvre lorsque l'on clique sur le titre.

Programmation :

- Définir chaque item avec son ***raccourci clavier obligatoire***.

Exemple :



Popup menu

Rôle :

- Proposer des items en dehors d'une barre de menu.

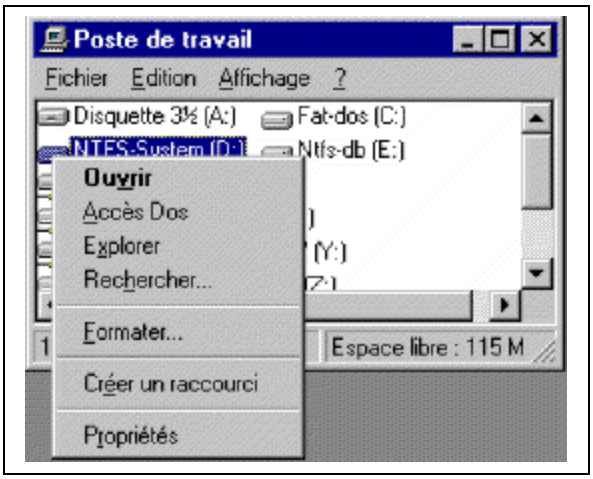
Caractéristiques :

- C'est un menu contextuel lié au focus de la souris. Il s'affiche d'ailleurs à l'endroit du pointeur souris.
- Accessible avec le bouton droit.

Programmation :

- Comme les menus déroulant.

Exemple :



Cascading menu

Rôle :

- Offrir un ensemble d'options supplémentaires à un item de menu.

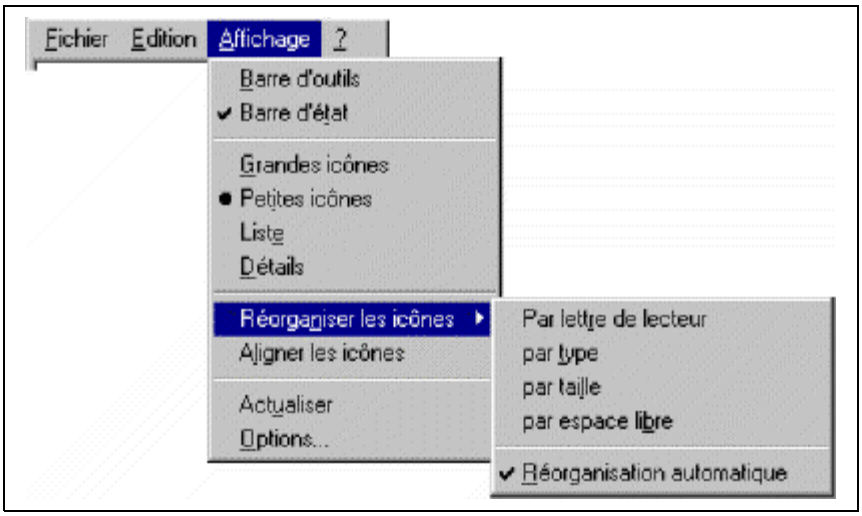
Caractéristiques :

- C'est un sous-menu. Il faut s'efforcer de ***limiter le nombre de niveaux à trois maximun***.

Programmation :

- Comme pour les menus déroulant.

Exemple :



Guide de style : l'essentiel en quelques règles

Introduction

Ce chapitre résume à lui tout seul, en 114 règles, la charte graphique 32 bits. Les règles y sont exposées de façon synthétique sous forme de résultats bruts, leurs fondements ayant été longuement exposés dans les chapitres précédents. Il est donc nécessaire, avant d'aborder ce chapitre, d'avoir lu (et compris) les précédents.

Règles générales relatives aux applications

Les règles inhérentes à Windows NT 4.0

Les ressources (couleurs et polices) doivent être les ressources systèmes. En effet, celles-ci sont devenues paramétrables sous Windows NT 4 et elles doivent être la référence des applications, ne serait ce que pour permettre les " options d'accessibilité " (telles les très grosses polices) de toujours fonctionner...

Règle 1	Le libellé dans la barre de titre d'une fenêtre ou d'un dialogue est toujours justifié à gauche.
Règle 2	La police du titre des fenêtres est la police système gras.
Règle 3	La police des menus et des icônes est la police système maigre.
Règle 4	Les raccourcis claviers standard sont: - **Ctrl +O**: Ouvrir, - **Ctrl +C**: Copier, - **Ctrl +X**: Couper, - **Ctrl +V**: Coller, - **Ctrl +Z**: Annuler, - **Ctrl +S**: Enregistrer, - **Ctrl +P**: Imprimer.
Règle 5	Les raccourcis clavier standards pour les fenêtres sont: - **F1**: affiche l'aide de l'application, - **Shift + F1**: active / désactive le mode d'aide contextuelle, - **Shift + F10**: affiche le menu pop up contextuel, - **Alt**: active la barre de menu de la fenêtre, - **Alt + Tab**: affiche la fenêtre primaire (ou l'application) suivante, - **Alt + Esc**: affiche la fenêtre suivante, - **Alt + F4**: fermeture de l'application ou d'un dialogue.
Règle 6	Les raccourcis clavier standard pour le MDI sont: - **Ctrl + F6**: affiche la fenêtre fille suivante, - **Ctrl + F4**: ferme la fenêtre fille active, - **Ctrl + W**: ferme la fenêtre fille active.
Règle 7	Le bouton " **OK** " est libellé en majuscules; son raccourci clavier est la touche " Enter " ou " Entrée ".

Règle 8	La bouton " **Annuler** " est libellé avec la première lettre en majuscule; son raccourci clavier est la touche " Escape ".
Règle 9	Un dialogue est modal sauf si c'est un dialogue de recherche.
Règle 10	Les fenêtres applicatives (principales ou filles MDI) sont non modales.
Règle 11	La touche **F1** est strictement réservée à l'appel de l'aide en ligne.

La typographie

Règle 12	Les items de menus qui provoquent l'affichage d'un dialogue sont suivis des trois points " ... ".
Règle 13	Le nom d'un item de menu ne doit pas avoir plus de 4 mots.
Règle 14	Les effets de reliefs sont à proscrire pour les libellés.
Règle 15	***Tous les textes et libellés de toute nature destinés à l'utilisateur doivent être exprimés en langue française.***

La couleur

Règle 16	La couleur de fond des dialogues est la couleur de fond de la palette système
Règle 17	Les contrôles indicateurs (type edit-box, list-box, etc.) ont toujours un fond blanc.
Règle 18	Les libellés typographiques sont obligatoirement de couleur noire. Ils sont en gris s'ils représentent un " inutilisable " (item de menu ou bouton).

Comportement des applications MDI et SDI

Règle 19	Une fenêtre principale MDI ou SDI ne contient qu'un seul et unique menu, qui peut éventuellement être géré dynamiquement selon le contexte.
Règle 20	La barre de menu d'une application MDI contient au minimum les items *Fichier*, *Edition*, *Fenêtre* et *" ? "*.
Règle 21	La barre de menu d'une application SDI contient au minimum les items *Fichier*, *Edition* et *" ? "*.

Situations modales et non modales

Règle 22	Il faut limiter les situations modales...
Règle 23	Il faut utiliser les onglets (classe des contrôles *Tabs*) plutôt qu'une succession de dialogues.
Règle 24	Un bouton dans un dialogue modal ne peut ouvrir qu'un autre dialogue.

Règles relatives aux fenêtres

Fenêtre principale

Règle 25	La fenêtre principale d'un application est déplaçable et redimensionnable. Elle contient obligatoirement les boutons système de fermeture, d'agrandissement et de réduction. Elle contient également obligatoirement une barre de titre et un icône d'application.
Règle 26	La barre de titre d'une application SDI contient l'icône de l'application, le nom de l'application et le nom de l'objet géré.
Règle 27	La barre de titre d'une application MDI contient l'icône de l'application, le nom de l'application et le nom de la fenêtre active si celle-ci est agrandie à la taille maximale.
Règle 28	Si la fenêtre principale est une boîte de dialogue (application type tableau de bord), alors elle est obligatoirement non modale.

Fenêtre fille

Règle 29	Le libellé de la barre de titre d'une fenêtre fille est toujours justifié à gauche.
Règle 30	Une fenêtre MDI fille a les mêmes attributs système que la fenêtre principale, sauf qu'elle n'est déplaçable qu'au sein de la fenêtre principale.
Règle 31	Une fenêtre fille MDI n'a pas de barre de menu, ni barre d'outil, ni barre d'état, ni bouton d'action. Tout doit être géré contextuellement au niveau de la fenêtre principale.
Règle 32	Une fenêtre fille MDI n'est jamais modale.

Dialogues

Règle 33	Un dialogue n'est jamais redimensionnable, et contient obligatoirement un bouton de fermeture, repris par le bouton système " **x** ".
Règle 34	Un dialogue ne doit pas contenir plus de 5 boutons.
Règle 35	Un bouton qui ouvre un sous-dialogue a un nom suivi des 3 points " ... ".
Règle 36	Un bouton qui agrandit un dialogue (qui en augmente le niveau de détails) a un nom suivi des 2 chevrons " >> ". Le retour à la taille initiale est assuré par un bouton dont le nom est précédé des chevrons inverses " << ".
Règle 37	Les boutons avec icône sont à réserver à la barre d'outils et non aux dialogues.
Règle 38	Le cadre d'un dialogue doit avoir un aspect rectangulaire, la longueur étant horizontale. Un rapport de 1,6 entre les deux dimensions offre un confort de lecture optimal.
Règle 39	Les cadres des contrôles sont en justification totale, tant horizontalement que verticalement.
Règle 40	Les regroupements de contrôles d'un même domaine se feront de préférence verticalement, plutôt qu'horizontalement.
Règle 41	Un bouton de validation ne peut qu'être libellé " **OK** ".
Règle 42	Les contrôles de validité d'une saisie doivent se faire, quand c'est possible, dès le déclenchement d'une action et non à la fin d'un dialogue...
Règle 43	... sinon, à la fin d'un dialogue erroné, il faut revenir à la fenêtre en positionnant le curseur souris et le focus sur le contrôle mis en cause, le tout accompagné d'un message d'erreur lui-même modal.
Règle 44	A l'opposé de la règle 42, la fermeture d'un dialogue par le bouton **Annulation** ne doit déclencher aucun contrôle de validité (et par-là même, aucun message d'erreur).
Règle 45	Un contrôle de saisie peut être rempli d'une valeur par défaut. Dans ce cas, ce doit être la dernière valeur renseignée précédemment par l'utilisateur.
Règle 46	Gabarit à respecter dans un dialogue (pour une résolution maximum de 800x600 pixels sur un moniteur de 15 pouces) : - Les espacements entre contrôles d'un même domaine sont de 6 pixels verticalement et 8 pixels horizontalement. - Les séparations, de 16 pixels verticalement et 18 pixels horizontalement. - Les marges avec les bords du cadre de la boîte de dialogue sont de 8 pixels verticalement et de 10 pixels horizontalement.

Dialogues standards

Règle 47	L'emploi des dialogues standards est obligatoire s'ils offrent les fonctionnalités recherchées. Ces dialogues sont au nombre de 7: *" Ouvrir "* pour rechercher et sélectionner un nom de fichier en parcourant l'arborescence. *" Enregistrer sous "* pour localiser un répertoire de l'arborescence où il faut sauvegarder un ficher. *" Rechercher "* pour effectuer des recherches de caractères dans un texte. *" Remplacer par "* pour effectuer des remplacements de caractères dans un texte. *" Imprimer "* pour définir les paramètres de l'impression. *" Choix d'une police "* pour sélectionner une police et définir les propriétés typographiques. *" Choix d'une couleur "* pour sélectionner des couleurs.

Barre d'outils (... rubans et règles)

Règle 48	La barre d'outils doit pouvoir être positionnée n'importe où dans la fenêtre et sur les bords haut, bas, gauche ou droit.
Règle 49	Si la barre d'outils est sur un bord, elle occupe en permanence toute la largeur ou hauteur de la fenêtre principale selon sa position.
Règle 50	Les boutons d'une barre d'outils sont obligatoirement des raccourcis représentant un item d'un menu.
Règle 51	Les boutons de la barre d'outils sont positionnés dans le même ordre que les items correspondants apparaissant dans le menu.
Règle 52	Les icônes standards de Windows NT 4.0 ne doivent être utilisés qu'avec les boutons dont les actions correspondent à celles prévues.
Règle 53	Il n'existe que deux tailles possibles pour les boutons d'un barre d'outil: - 24x22 pixels avec des icônes de 16x16 pixels, - 32x30 pixels avec des icônes de 24x24 pixels.
Règle 54	La barre d'outil est escamotable et peut être mise sous forme de palette.

Barre d'état

Règle 55	La barre d'état doit être escamotable et donc ne pas être indispensable.
Règle 56	Elle est sur le bord inférieur et occupe toute la longueur de la fenêtre.
Règle 57	Une aide contextuelle sur chaque menu et item doit y être placée.
Règle 58	Les messages de la barre d'état ne peuvent occuper qu'une seule ligne.

Règles relatives aux menus

Barre de menu

Règle 59	Le menu ***Fichier*** est obligatoire et contient au minimum l'item ***Quitter***.
Règle 60	Le menu ***Edition*** est réservé à la gestion du presse-papiers.
Règle 61	Le menu ***Fenêtre*** est réservé aux applications MDI.
Règle 62	Le nombre de menus d'une barre de menu ne doit pas dépasser 7.
Règle 63	Le libellé du menu est toujours en un seul mot dans la barre de menu.
Règle 64	Un menu dont tous les items sont inactifs doit être inactif (et donc grisé), ou mieux: ne pas être affiché.
Règle 65	Tous les menus ont obligatoirement un raccourci clavier ***Alt+lettre soulignée***

Menus déroulants (...et en cascade)

Règle 66	Si le menu Edition est présent il contient obligatoirement les items suivants: ***- Annuler Ctrl + Z*** ***– Couper Ctrl + X*** ***– Copier Ctrl + C*** ***– Coller Ctrl + V***
Règle 67	Si le menu Fenêtre est présent, il contient obligatoirement les items suivants: ***- Cascade*** ***– Mosaïque*** ***– Réorganiser*** ***(liste des fenêtres)***
Règle 68	Le menu d'aide contient au minimum l'item ***A propos***... pour obtenir les numéros de version et de série de l'application et éventuellement la licence.
Règle 69	Si un item de menu sert également d'indicateur, alors il faut également prévoir la marque d'état (actif / inactif) sur sa gauche.
Règle 70	Le nombre de (sous) items dans un (sous) menu est limité à 7.
Règle 71	Le nombre de niveaux d'imbrications dans un menu en cascade est limité à 2.
Règle 72	Il ne faut pas placer un item isolé entre deux barres de séparation.
Règle 73	Un item indisponible n'est pas affiché, ou alors il est éventuellement grisé.

Pop up menus

Règle 74	Les items des menus pop up sont soumis aux mêmes règles que ceux des menus déroulant (règles 69 à 73).
Règle 75	Les items d'un menu pop up doivent être également disponibles depuis le menu déroulant.

Nom et items de menu

Règle 76	Les noms des menus sont limités à un seul et unique mot, écrit en minuscules avec la première lettre en majuscule.
Règle 76	Les nom des items de menus sont limités à 4 mots, dont seul le premier voit sa première lettre en majuscule.
Règle 77	Un menu sans item est suivit du point d'exclamation.
Règle 78	Les raccourcis claviers standard pour les menus sont: - " Alt + F " pour *Fichier*, - " Alt + E " pour *Editer*, - " Alt + n " pour *Fenêtre*, - " Alt + ? " pour l'aide.

Règles relatives aux contrôles

Bouton d'action

Règle 79	Dans un dialogue avec onglets, les boutons communs sont en dehors des onglets alors que les boutons relatifs à un onglet sont positionnés dessus.
Règle 80	Les boutons ont toujours un effet 3D et trois états (relâché, enfoncé et indisponible ou grisé). Leur couleur de fond est le gris clair.
Règle 81	La taille des boutons est unique dans un dialogue donné. Au pire, les boutons d'un même domaine doivent être de dimensions identiques.
Règle 82	La largeur des boutons sans image est de 72 pixels min. et 100 pixels max.
Règle 83	Les boutons d'actions doivent être accessibles par des raccourcis clavier du genre *Alt + lettre soulignée.*
Règle 84	Le bouton d'aide contextuelle est libellé *"Aide"*. Le "**?**" est réservé au menu.
Règle 85	Le bouton d'aide contextuelle est toujours placé en dernier dans la liste des boutons disponibles.
Règle 86	Le libellé " *Fermer "* est réservé au bouton mettant fin à un dialogue de consultation.

Bouton radio et case à cocher

Règle 87	Le libellé d'un contrôle dont le contenu est modifiable comporte obligatoirement un raccourci clavier.
Règle 88	La couleur de fond des cases à cocher est le blanc lorsqu'elle est modifiable et le gris lorsqu'elle ne l'est pas.

Boîte de regroupement

Règle 89	Un libellé associé au contrôle *group-box* est affiché à gauche et comprend un espace avant et après le texte. Les effets 3D à cet endroit sont proscrits.

Liste de choix, drop down et combo box

Règle 90	La taille des indicateurs doit être proportionnelle à celle des informations qu'ils contiennent.
Règle 91	Les contrôles *list-box*, *combo-box*, *drop-down-list* et *spin-box* ont un fond de couleur blanche. Ils possèdent obligatoirement un cadre.
Règle 92	Une *list-box* doit permettre d'afficher entre 3 éléments minimum et 7 éléments maximum.
Règle 93	Une *list-box* multi-colonnes avec des informations différentes doit obligatoirement avoir un en-tête de colonne.
Règle 94	Une *list-box* contient : - soit des icônes 32x32 seuls, - soit des icônes 16x16 avec un texte.
Règle 95	Si le choix proposé par un *drop-down list* est facultatif, ce contrôle comporte obligatoirement un élément " aucun " en première position.
Règle 96	Les contrôles *combo-box* et *drop-down list* n'ont jamais d'en-tête.
Règle 97	Le contenu des indicateurs est toujours justifié à gauche.

Champ de saisie, d'affichage et libellé

Règle 98	Les effets de relief sont proscrits pour les libellés.
Règle 99	Un champ d'affichage a un fond de la couleur du dialogue et pas de cadre.
Règle 100	Un libellé qualifiant un objet inactif doit être grisé.
Règle 101	Un libellé qualifiant un contrôle dont le contenu est modifiable comporte obligatoirement un raccourci clavier.
Règle 102	A l'inverse de la règle 101, un libellé qualifiant un indicateur dont le contenu n'est pas modifiable n'a en aucun cas de raccourci clavier.
Règle 103	Un libellé qualifiant un champ de saisie monoligne est obligatoirement aligné avec lui horizontalement et placé à sa gauche.
Règle 104	Un libellé qualifiant un champ de saisie multi-lignes est obligatoirement aligné avec lui verticalement et placé au-dessus.

Les onglets

Règle 105	Les index d'onglet sont alignés à gauche sur le fond de l'onglet et accolés les uns aux autres. Leur titre est justifié à gauche.
Règle 106	Les index d'onglet ont obligatoirement un effet 3D et une couleur de fond identique à celle du dialogue.

Messages et assistance aux utilisateurs

Règle 107	Les bulles d'aide doivent être supportées, pour les boutons d'une barre d'outils, pour les images d'une liste de choix, pour les index d'onglets.
Règle 108	Le message d'information est utilisé pour informer l'utilisateur du résultat de son action. Il est caractérisé de façon indissociable par: une situation modale, l'icône " **i** ", un bouton " **OK** ".
Règle 109	Le message d'alerte est utilisé pour notifier à l'utilisateur que la situation requiert une décision due aux risques de pertes d'informations qu'elle peut entraîner. Il est caractérisé de façon indissociable par: une situation modale, l'icône " **!** ", les boutons " **OK** " ou " **Oui** " et " **Annuler** " ou " **Non** ".
Règle 110	Le message d'erreur grave est utilisé pour indiquer que l'action ne peut s'exécuter suite à une erreur système. Il est caractérisé de façon indissociable par: - une situation modale, - l'icône " **x** ", - soit le bouton " **OK** " si l'abandon est la seule action possible, - soit les boutons " **Répéter** " et " **Annuler** " s'il y a alternative.
Règle 111	Le pointeur doit être transformé en sablier si l'action lancée empêche toute autre action au sein de l'application.
Règle 112	Le pointeur doit être transformé en sablier et pointeur si l'action lancée ne bloque pas le reste de l'application *(multi-threaded application)*.
Règle 113	Si une action dure plus de 5 secondes, l'indicateur de progression doit être obligatoirement utilisé et maintenu en avant plan, que l'action soit bloquante ou non pour l'application. L'interruption de l'action en cours doit toujours être possible avec un bouton " **Abandonner** ".
Règle 114	Seul le bip système peut être utilisé comme message sonore, message sonore qui ne peut en aucun cas constituer le seul élément de retour d'information.

Portage des applications 3.11 vers NT4.0

Introduction

Avant d'aborder les points divergents entre ces deux systèmes d'exploitation, il convient de faire une remarque: Windows NT 4.0 est parfaitement capable d'exécuter directement des applications 16 bits développées auparavant pour Windows 3.1 sous deux conditions:

1. ***la plate-forme NT soit bien sûr autour d'une architecture x86,***
2. ***les éléments de l'interface graphique soient des standards de Windows 3.1.***

L'émulation est particulièrement complète puisqu'elle va jusqu'à la manipulation des variables systèmes internes à Windows 3.1. Cependant, avec le début de la convergence de Windows 95 et de Windows NT 4.0 vers un même système d'exploitation, il n'est pas sûr que cette compatibilité soit maintenue dans l'avenir, et ce dès les prochaines versions (Windows-98 - ex " 97 " - et Windows NT 5). De plus, Windows NT est disponible sur des architectures autres que PC...

Envisager une recompilation est donc à l'ordre du jour. C'est alors l'occasion de mettre l'application aux nouvelles normes de développement imposées par les caractéristiques de Windows NT. Mais il ne faut pas se méprendre, selon la qualité de la programmation de l'application sous Windows 3.11, le portage peut soit être immédiat avec des modifications mineures, soit tout simplement impossible...

Différences techniques entre les deux interfaces graphiques

Concernant l'aspect visuel

Les différences sont nombreuses (effets 3D sur les rebords des fenêtres, boutons systèmes refondus, suppression des icônes des fenêtres réduites...) mais restent entièrement prises en compte par la compatibilité ascendante du système *pour peu que l'on ait pris soin de développer avec le SDK de Windows 3.1*.

Le seul travail qui reste à faire est minime puisqu'il ne concerne que des justifications de texte à gauche, des changements de polices... Et ceci, seulement si l'on veut à tout prix se plier aux nouveaux concepts de Windows NT 4.0. Dans le cas contraire, une simple recompilation suffira.

Concernant l'adressage mémoire

Cette recompilation est d'ailleurs souhaitable, même si la plate-forme NT est toujours du type x86, même si l'on ne tient pas particulièrement à passer aux normes de Windows NT 4.0. La raison en est simple: Windows NT est un OS 32 bits, alors que Windows 3.1 est 16 bits, ce qui explique que le maintien de l'exécutable 16 bits pose un gros problème d'adressage mémoire.

En effet, il faut traduire le mode d'adressage segment - offset 16-16 en mode linéaire 0-32 *dynamiquement*, c'est à dire pendant l'exécution de l'application et de ses DLL (ce processus porte le nom de *thunking*). Si on sait que cette traduction d'adresses est parfaitement réussie et maîtrisée chez OS/2 Warp 3 d'IBM, on sait en revanche qu'elle est complètement inexistante chez Windows 95. Pour Windows NT 4.0, on ne sait pas comment cela est réalisé (Microsoft étant particulièrement avare en détail technique), on a donc toutes les raisons de ne pas faire confiance à ce mécanisme. En recompilant, donc, l'ancienne application passera en 32 bits et là on peut être sûr que la robustesse de Windows NT 4.0 ne sera jamais mise en défaut.

Concernant les informations de configuration

Là, les différences sont fondamentales: le fichier *.ini* est mort avec l'arrivée de Windows NT 4.0. C'est à présent dans la *base des registres* qu'il faut aller stocker les informations de paramétrage et de configuration, tant du système que des applications. Au titre de la compatibilité, Windows NT 4.0 a cependant maintenu l'existence de *Win.ini*, *System.ini* et *Winfile.ini*. Toutefois, installer une application 16 bits sous Windows NT 4.0 qui va modifier ces fichiers est absolument sans effet sur le nouveau système d'exploitation, puisque les modifications de ces fichiers ne sont pas reportées dans la base de registres.

Le travail d'adaptation pour passer de Windows 3.11 à Windows NT 4.0, risque donc d'être plus conséquent si l'on tient à tout prix à ce que ce soit l'application qui fasse plier l'OS à ses exigences de fonctionnement, plutôt que l'inverse.

Concernant les échanges de données entre les applications

Avec Windows 3.1x, il existe essentiellement trois modes d'échange de données entre les applications: le presse-papiers, le DDE (Dynamic Data Exchange), et OLE (Object Linking & Embedding). Le presse-papiers est géré de la même manière sur les deux systèmes et il n'y aura rien à faire pour le portage, toujours sous réserve d'avoir développé avec le SDK de Windows, ou tout du moins d'avoir scrupuleusement utilisé l'API.

Le DDE est un canal d'échange de données ouvert entre deux applications. Si c'est le plus utilisé sous Windows 3, c'est parce qu'il est très simple à mettre en œuvre. Il est parfaitement soutenu par Windows NT 4.0 pour des raisons de compatibilité ascendante, mais son avenir est très compromis car Microsoft a tout misé sur son successeur: l'OLE. Si l'application devait survivre encore longtemps après Windows NT 4.0, il vaudrait mieux dès à présent songer, pour le portage, à abandonner la technologie DDE. Le travail risque d'être très lourd dans ce cas.

L'OLE est le tournant stratégique de Microsoft. Il en est déjà à sa version 2. C'est plus qu'un moyen d'échange de données: il permet l'imbrication d'objets entre deux applications. C'est sur cette technologie que s'appuie l'OLE Control Architecture, permettant de réaliser des briques de fonctionnalités logicielles de façon beaucoup plus générale que ne le permettait le DDE. Ces éléments, les OCX (*OLE Custum Controls*) ont déjà détrôné les VBX de Visual Basic et sont désormais récupérables par tous les L4G. Ainsi, si l'IHM utilise des composants OCX, il n'y aura aucun travail à faire, autre que la recompilation, pour obtenir une nouvelle mouture de l'application qui traversera toutes les versions suivantes de Windows NT.

Opportunité d'un portage vers Windows NT

Comme on vient juste de le voir, le travail à faire pour porter une application vers le nouvel environnement graphique peut être soit dérisoire soit démesuré. Il convient donc de bien étudier l'opportunité de ce portage avant d'en prendre la décision. Il n'y a en fait que deux questions fondamentales à se poser:

1. La pérennité de l'application dépassera-t-elle la durée de vie de Windows NT 4.0 ? Compte tenu des objectifs avoués par Microsoft en ce qui concerne ses OS, on ne peut que s'en tenir à Windows NT 4.0 pour le moment comme garant d'un maintien d'une compatibilité ascendante avec Windows 3.11. Si l'application n'est pas destinée à survivre à Windows NT 4.0, il n'y a aucune raison de considérer un portage comme nécessaire à sa pérennité.

2. L'application a-t-elle été proprement écrite ? C'est à dire, est-ce qu'il n'est fait appel qu'au SDK de Windows 3.11 pour la construction de l'IHM. C'est une question à bien explorer car il existe beaucoup d'AGL de développement qui proposent leur propre collection de contrôles pour construire l'interface graphique de l'application. Or, ces contrôles ne sont bien souvent livrés que sous forme d'exécutables précompilés en bibliothèque (pour des raisons commerciales évidentes) et sans les sources, il n'y a aucune recompilation possible, donc pas de portage envisageable.

Portabilité des développements actuels

La question du portage de Windows 3.11 à Windows NT 4.0 se pose également dès aujourd'hui pour les applications en cours de développement. Leur cible reste Windows 3.11 puisque que c'est encore la référence des postes de travail du parc installé. Pour garantir un portage futur sans problème, la solution a été apportée depuis longtemps par Microsoft avec les extensions Win32s.
Il est donc déjà possible de développer des applications à travail de portage futur nul pour Windows NT 4.0 si l'on utilise un L4G tel Delphi 2.0 avec l'extension Win32s sur un poste 16 bits. A l'installation du logiciel développé, soit on installe la bibliothèque Win32s si le poste cible est Windows 3.11, soit on ne l'installe pas si la cible est Windows NT 4.0. Un L4G tel que celui précité est la garantie d'une portabilité totale entre ces deux systèmes puisque les contrôles respectent la norme OLE 2, en plus d'un respect de l'API de Windows. Le passage par une bibliothèque de transition entre les deux systèmes d'exploitation pourrait mener à considérer ce processus comme du bricolage, mais il n'en est rien. D'ailleurs, Microsoft lui-même à bien précisé que les applications développées pour Windows NT 4.0 et Windows 95 ne pouvaient être qualifiées du label *" Designed for Windows 95 "* que si elles ne reposaient que sur l'API Win32s...

Grille d'évaluation d'une IHM lors du maquettage

Introduction

Cette grille, par un jeu de questions - réponses, permet de mesurer rapidement la conformité d'une IHM aux règles exposées dans cette charte graphique. Elle ne s'appuie que sur le "minimum minimorum" des règles à respecter. Cette grille de tests ne peut donc à ce titre constituer un résumé de la charte graphique, dont il faut entièrement tenir compte pour le développement d'interfaces homme - machine.

Le mode d'emploi est le suivant: chaque question doit être répondue par l'affirmative. Dans ce cas, les cases sont cochées. En cas de réponse négative, la case correspondante reste vierge. Il apparaît ainsi du premier coup d'œil les endroits où l'IHM pêche. Le questionnaire étant un minimum minimorum, le fait de ne pas cocher une seule case (donc une seule réponse négative) est suffisant pour entraîner le refus de l'IHM proposée.

Aspect général

L'application contient-elle un seul menu ?	
L'application comporte-t-elle moins de trois niveaux de profondeur ?	
L'application est-elle sobre (i.e. très peu de couleurs) ?	
L'application possède-t-elle une aide en ligne ?	
L'application est-elle MDI si c'est une application de gestion ?	
Les libellés sont-ils tous exprimés en langue française ?	

Fenêtre principale

Est-ce que la barre d'outils (si elle existe) prend toute la largeur de la fenêtre ?	
Les boutons de la barre d'outils ont-ils tous les mêmes dimensions ?	
Les fonctions de la barre d'outils sont-elles également disponibles depuis les menus ?	
Est-ce que la fenêtre ne contient pas de boutons d'action ?	
La barre d'état (si elle existe) prend-elle toute la largeur de la fenêtre ?	
La barre d'état est-elle escamotable ?	

Les menus

La barre de menu contient-elle sept menus maximum ?	
Si l'application est SDI, a t-on les menus *Fichier, Edition* et *Aide ?*	
Si l'application est MDI, a t-on les menus *Fichier, Edition, Fenêtre* et *Aide ?*	
Les menus sont-ils en un seul mot ?	
Les menus comportent-ils au minimum deux items ?	
Les items sont-ils regroupés par un maximum de sept ?	
Le menu *Fichier* contient-il l'item *Quitter* ?	
Le menu *Edition* contient-il les items *Annuler, Couper, Copier,* et *Coller* ?	
Le menu *Fenêtre* contient-il les items *Mosaïque, Cascade* et *Réorganiser* ?	
Le menu *?* Est-il le dernier de la barre de menu ?	
Le menu *?* Contient-il au moins l'item *A propos* ?	
Un équivalent clavier est-il associé à chaque menu ?	
Les menus en cascade sont-ils limités à deux niveaux ?	
Les items ouvrant un dialogue sont-ils suivis de "... " ?	

Les menus pop up (s'ils existent) sont-ils contextuels ?	
Les items des pop up sont-ils accessibles depuis la barre de menu ?	

Les fenêtres filles

Les filles MDI sont-elles non modales, redimensionnables et icônifiables ?	
Les filles MDI sont-elles liées à l'espace de la fenêtre principale ?	
Les filles MDI sont-elles sans barre de menu ?	
Les filles MDI sont-elles sans barre d'outils ?	
Les filles MDI sont-elles sans bouton ?	
Les filles MDI sont-elles sans barre d'état ?	

Dialogues

Les dialogues sont-ils non redimensionnables ?	
S'il ne s'agit pas de dialogues de recherche, sont-ils tous modaux ?	
La taille des dialogues est-elle adaptée au contenu, pour toute résolution d'écran ?	
La couleur pour tous les dialogues est-elle unique ?	
A-t-on une justification totale des contrôles au sein des dialogues ?	
Les dialogues de consultation ont-ils le bouton *Fermer* ?	
Les autres ont-ils les boutons *OK* et *Annuler* ?	
Les dialogues ont-ils au maximum 7 boutons ?	
Sinon, ont-ils 7 boutons visibles max. et un bouton d'extension " >> " ?	
Les contrôles sont-ils regroupés par fonctionnalités ?	

Les fonctions standards se font-elles à travers les dialogues standards ?	

Contrôles

Tous les boutons *" OK "* sont-ils libellés en majuscule ?	
Les boutons *" OK "* ont-ils tous à gauche ou au-dessus des boutons *Annuler* ?	
Les boutons *" OK "* ont-ils tous la touche *Enter* comme raccourci clavier ?	
Les boutons *Annuler* ont-ils la touche *Esc* comme raccourci clavier ?	
Les libellés des boutons ont-ils la première lettre en majuscule ?	
Un équivalent clavier est-il donné à chaque bouton ?	
Passe-t-on d'un contrôle au suivant par la touche *Tabulation* ?	
Passe-t-on d'un contrôle au précédent par les touches *Shift + Tabulation* ?	
Les champs d'affichage modifiables sont-ils sur fond blanc ?	
Les boutons radios sont-ils regroupés dans une boîte de regroupement ?	
Chaque contrôle a-t-il un libellé ?	
Un libellé associé à un champ de saisie a-t-il un raccourci clavier ?	
Un libellé associé à un champ d'affichage est-il sans raccourci clavier ?	
Le contenu des contrôles est-il justifié à gauche ?	

Messages

Les messages d'information comportent-ils le bouton *OK* et l'icône " **i** " **?**	
Les messages d'avertissement ont-ils les boutons *OK* ou *Répéter* et *Annuler* et l'icône " **!** "	
Les messages d'erreur grave ont-ils le bouton *OK* et l'icône " **x** " ?	
L'application affiche-t-elle le sablier pour des traitements de 1 à 5 secondes ?	
L'application affiche-t-elle la barre de progression pendant un traitement de plus de 5 secondes ?	
Est-il possible d'arrêter un traitement long à tout moment, par un bouton *Annuler* présent en même temps que la barre de progression ?	