Delphi – Partie 2
Les Composants Fondamentaux

Ecrit par Jean-Paul Pruniaux

Objectif

Pouvoir réaliser des applications avec *Delphi* en utilisant les composants fondamentaux d'interfaçage avec *Windows*. Ce cours est le deuxième d'une série de cours destinés à former des programmeurs *Delphi*.

Niveau requis

L'étudiant doit avoir fait le cours « *Delphi – Partie 1 — Le Pascal Objet* ».

Durée

3 jours.

Droits d'auteur

TABLE DES MATIERES

1. INTRODUCTION A LA PROGRAMMATION SOUS DELPHI

Commencez par démarrer *Delphi* en sélectionnant dans le menu système de *Windows* la commande suivante : « *Démarrer / Programmes / Borland Delphi4 / Delphi 4* » ou en cliquant sur l'icône adéquate. L'écran doit prendre une apparence qui ressemble à ceci :

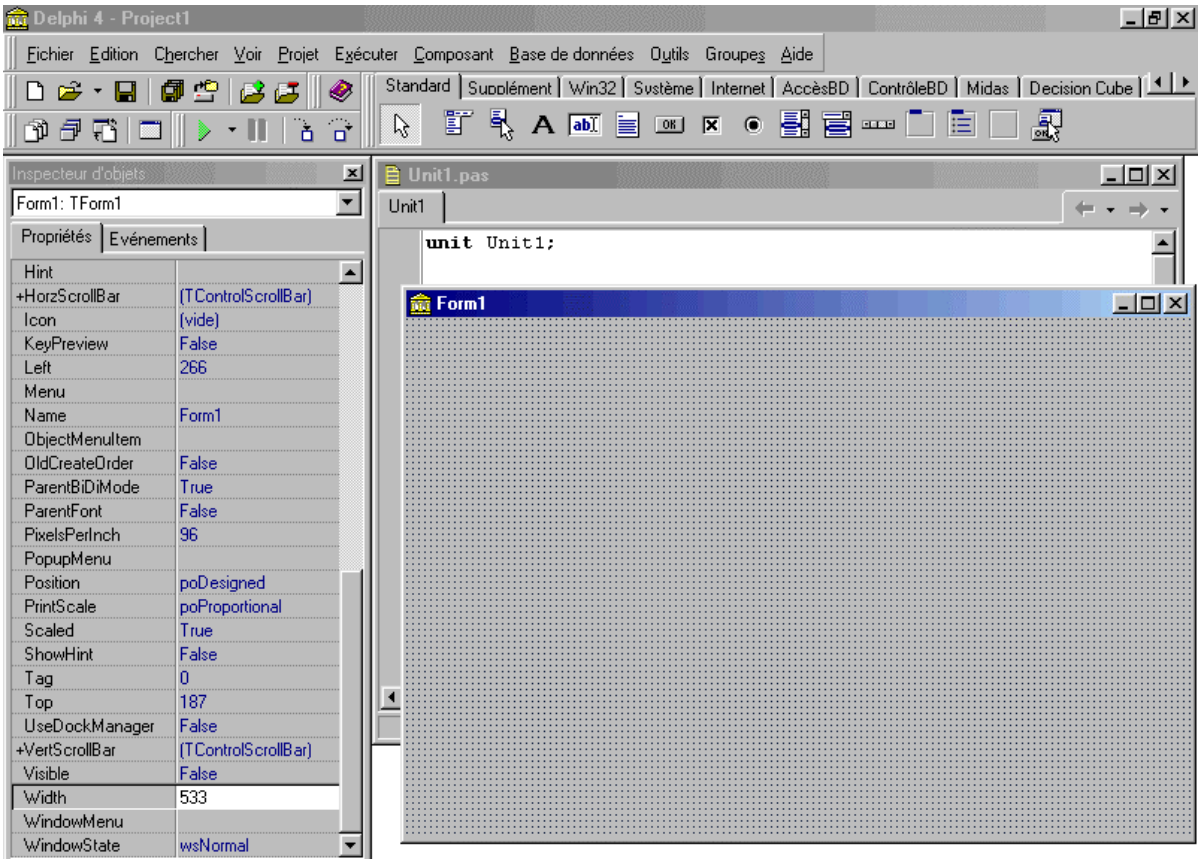

Figure 1. L'environnement de travail de Delphi

Remarque

Lorsque vous démarrez *Delphi*, l'apparence de l'écran peut changer en fonction de la configuration sur votre machine et en fonction des derniers travaux réalisés avec *Delphi*.

Il peut arriver que Delphi vous présente la dernière application en cours de conception — Auquel cas, utilisez la commande « *Fichier / Nouvelle application* » pour démarrer un nouveau projet.

En haut de l'écran se trouve la barre d'outils de Delphi, outils principal de son utilisation. Cette barre d'outils comporte les éléments suivants :

- Un menu rassemblant toutes les commandes possibles.
- Sous le menu à gauche, une barre d'outils avec les icônes servant de raccourci pour les commandes les plus fréquemment utilisés.
- Sous le menu à droite, la *palette de composants*. Celle-ci constitués d'onglets (« *Standard* », « *Suppléments* », « *Win32* »…) permet d'insérer facilement n'importe quel composant graphique de la bibliothèque de Delphi dans une fenêtre de programme.

Remarque

De nombreux menus contextuels, accessibles par le bouton gauche de la souris permettent d'envoyer des commandes particulières à un objet de l'environnement de travail.

Sur la gauche de cet écran se trouve l'*inspecteur d'objet*. Celui-ci permettra de paramétrer n'importe quel composant inséré dans une fenêtre en cours de développement.

Sur la droite de cet écran se trouvent deux fenêtres « *Form1* » et « *Unit1.pas* » :

- La fenêtre actuellement appelée « *Form1* » est destinée à concevoir graphiquement le contenu de la fenêtre principale de la nouvelle application. Dans cette fenêtre vous pouvez insérer des composants à partir de la *palette de composants*.
- Le fenêtre actuellement appelée « *Unit1.pas* » contient des éditeurs de texte destinés à concevoir ou modifier le source du programme et de ses unités.

1.1 RAPPEL SUR LES COMMANDES FONDAMENTALES

1.1.1 La sauvegarde des sources

Avant d'exécuter une application, nous vous engageons à sauvegarder tous les éléments constituant le projet :

Utilisez si nécessaire la commande « *Fichier / Tout enregistrer* » pour enregistrer tous les modules sources du projet et son paramétrage.

Remarque

Afin de faciliter la gestion de l'ordinateur avec lequel vous travaillez pendant cette formation, veuillez créer un répertoire à part et y stocker tous les éléments que vous développer.

Remarque

Par défaut, un projet est constitué de deux éléments principaux :

- Un premier source dont le nom par défaut est « *Unit1* » contenant la définition et le paramétrage de la fenêtre principale — Ce source est en fait constitué de 2 fichiers : « *Unit1.pas* » (le source *Pascal*[1]) et « *Unit1.dfm* » (le paramétrage de la fenêtre principale[2]).
- Un deuxième source dont le nom par défaut est « *Project1.dpr*[3] » contenant le programme principal. Il s'agit d'un source Pascal commençant par l'instruction « `program` ».

[1] Un source Pascal est un fichier texte contenant des instructions Pascal – Une unité Pascal comporte l'extension « *pas* ».

[2] « *DFM* » signifie « *Delphi ForM* » ou « *Fiche Delphi* » — Un fichier d'extension « *DFM* » contient toutes les valeurs par défaut permettant d'initialiser correctement une fenêtre lors de sa création à l'exécution.

[3] « *DPR* » signifie « *Delphi PRoject* » ou « *Projet Delphi* » — Un fichier d'extension « *DPR* » est un source Pascal contenant le programme principal d'une application *Delphi*.

1.1.2 La compilation et l'exécution d'une application

Pour compiler et exécuter le programme, utilisez la commande « Exécuter / Exécuter », appuyez sur la touche F9, ou cliquez sur l'icône ci-contre. 

Delphi vérifie alors tous les sources du projet et compile ceux-ci si la date du source est plus récente que la date du module compilé.

Delphi compile alors le source principal de l'application puis édite les liens.

Finalement *Delphi* lance l'application. Sa fenêtre principale s'ouvre et vous pouvez tester le programme.

Remarque

La commande « *Projet / Compiler le projet*[4] » vérifie quels sources ont besoin d'être compilés, puis compile le programme principal et édite les liens (réalisation des deux premières étapes précédentes).

La commande « *Projet / Construire le projet* » compile tous les sources du projet et édite les liens.

Delphi 4 permet d'ouvrir plusieurs projets en même temps ; les commandes « *Projet / Compiler tous les projets* » et « *Projet / Construire tous les projets* » s'appliquent de la même manière à tous les projets chargés.

1.1.3 Naviguer dans Delphi

Delphi gère un projet au travers de diverses fenêtres, il faut très rapidement savoir passer d'une fenêtre à une autre et notamment faire apparaître la fenêtre dont on a besoin :

La commande « *Voir / Basculer fiche/unité*[5] » permettent de faire l'aller retour entre une fenêtre en cours de construction et le source correspondant.

La commande « *Voir / Inspecteur d'objets*[6] » permettent de faire l'aller retour entre une fenêtre, l'*inspecteur d'objet* et le source correspondant.

La commande « *Voir / Fiche*[7] » montre la liste de toutes les fiches constituant un projet.

La commande « *Voir / Unité*[8] » montre la liste de tous les sources *Pascal* constituant un projet.

La commande « *Voir / Gestionnaire de projet* » montre dans une nouvelle fenêtre tous les éléments constituant les projets chargés. Ceux-ci sont classés logiquement. *Delphi* peut stocker un environnement de travail dans un fichier « *Groupe de projets* » (*Project Group*) — Ceci explique le premier item de la fenêtre : « *ProjectGroup1* ».

---

[4] Cette commande est aussi accessible avec les touches Ctrl F9.

[5] Cette commande est aussi accessible avec la touche F12.

[6] Cette commande est aussi accessible avec la touche F11 .

[7] Cette commande est aussi accessible avec les touches ⇧ Shift F12.

[8] Cette commande est aussi accessible avec les touches Ctrl F12.

1.1.4 Placer des composants dans une fenêtre

Pour placer des composants dans une fenêtre, commencez par mettre cette fenêtre en avant-plan en utilisant les commandes de la section précédente.

Sélectionnez l'onglet adéquat dans la *palette de composants*, cliquez sur le composant désiré, puis cliquez dans la fiche.

Eventuellement, lorsque vous cliquez dans la fiche vous pouvez tracer le rectangle d'encombrement du composant inséré.

Par exemple pour insérer un bouton :

- Mettez la fenêtre en avant plan.
- Cliquez sur l'onglet « *Standard* » de la *palette de composants*.
- Cliquez sur l'icône  .
- Cliquez en haut à gauche de la fiche et maintenez le bouton de la souris enfoncé.
- Déplacez la souris un peu en bas à droite.
- Relâchez le bouton de la souris.
- Basculez vers l'*inspecteur d'objet*.
- Modifiez la propriété « *Caption*[9] » de ce bouton pour afficher « *Fermer* ».

Remarque

Vous pouvez insérer plusieurs fois le même composant dans une fiche ; pour ceci appuyez sur la touche  en même temps que vous cliquez sur l'icône du composant à insérer. Puis vous cliquez à plusieurs endroit de la fiche.

Lorsque tous les composants sont insérés, cliquez sur l'icône  .

Remarque

Certains composants ne sont visible que lors de la conception de la fiche et ne le seront plus à l'exécution du programme. Il s'agit de composants qui ont été conçu pour paramétrer facilement des fonctionnalités « invisibles ». Nous trouvons par exemple tous les composants de l'onglet « *AccèsBD* » permettant d'accéder aux bases de données.

D'autres composants sont visibles aussi bien en conception qu'à l'exécution. Nous parlons alors de *contrôles*. Nous trouvons par exemple des boutons, des cases à cocher, des zones de saisie…

Une fois qu'un composant a été posé sur une fiche, vous pouvez cliquer sur celui-ci pour le sélectionner. Vous pourrez alors le déplacer ou en modifier la taille grâce aux opérations classiques de travail à la souris sous *Windows*.

Avant de paramétrer un composant avec l'inspecteur d'objet, vous devez le sélectionner.

[9] « *Caption* » pourrait se traduire par « *Titre* », « *Sous-titre* » ou « *Légende* »

1.1.5 Les outils d'alignement

Plusieurs outils sont disponibles pour vous permettre d'aligner proprement les composant sur une fiche :

La grille magnétique

En conception, une fiche présente une « *grille magnétique* » représentée par une trame de points sur la fiche. Lorsque vous déposez ou déplacez un composant sur une fiche, la grille peut attirer automatiquement les coordonnées du composant. La commande « *Outil / Options d'environnement* » montre une boîte de dialogue dont l'onglet « *Préférences* » permet de gérer la grille magnétique :

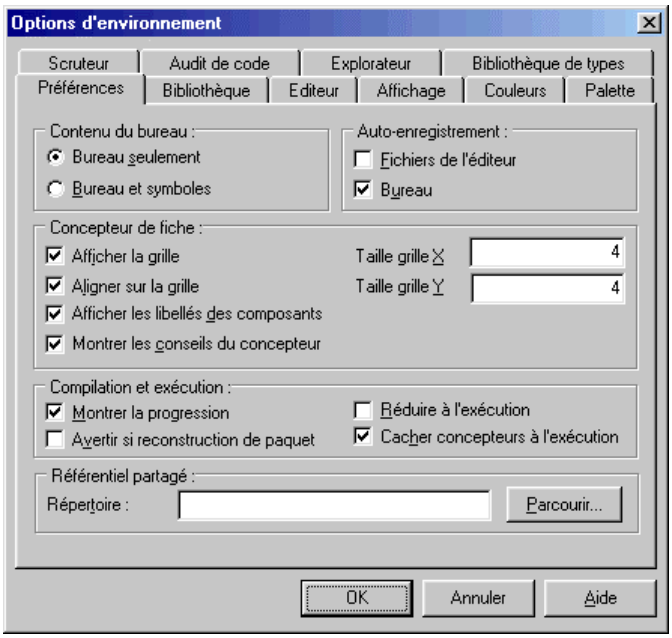

Figure 2. Le paramétrage de la grille magnétique

Les paramètres « *Concepteur de fiche* » permettent de gérer la « *grille magnétique* » :

- « *Afficher la grille* » : lorsque cette case est cochée, la grille est visible.
- « *Aligner sur la grille* » : lorsque cette case est cochée, la grille est « magnétique » et attire les composants lors de leur déplacement.
- « *Taille grille X* » et « *Taille grille Y* » déterminent le pas horizontal et vertical de la grille.

Nous vous engageons à étudier les autres paramètres de cet onglet « *Préférences* » dans le système d'aide de Delphi (Cliquez sur le bouton « *Aide* » ou appuyez sur la touche F1).

Les outils d'alignements

Pour aligner plusieurs composants les uns par rapports aux autres, vous pouvez utiliser divers outils d'alignement. Pour ceci :

- Sélectionnez tous les composants à aligner (Cliquez sur le premier, puis tout en maintenant la touche ⇧ Shift enfoncée, cliquez sur les suivants)[10].
- Avec le bouton droit de la souris, cliquez sur un de ces composants. Ceci fait apparaître un menu contextuel comprenant entre autres des commandes d'alignement :

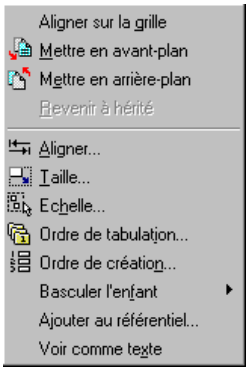

Figure 3. Le menu contextuel des composants

- La commande « *Aligner sur la grille* » force l'alignement des composants sélectionnés sur la grille.
- La commande « *Aligner* » fait apparaître une boîte de dialogue permettant de gérer l'alignement des composants les uns par rapport aux autres :

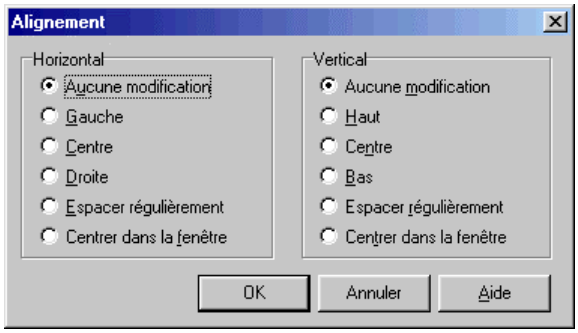

Figure 4. L'alignement des composants les uns par rapport aux autres

Si vous le souhaitez, vous pouvez consulter l'aide pour comprendre la signification des options d'alignement (Cliquez sur « *Aide* » ou appuyez sur F1). Choisissez une option d'alignement horizontal et une option d'alignement vertical puis validez la boîte de dialogue.

[10] Si les composants sont directement posés sur la fiche, vous pouvez aussi tracer avec la souris le rectangle englobant tous les composants que vous désirez sélectionner.

- La commande « *Taille* » fait apparaître une autre boîte de dialogue permettant d'ajuster la taille respective des composants :

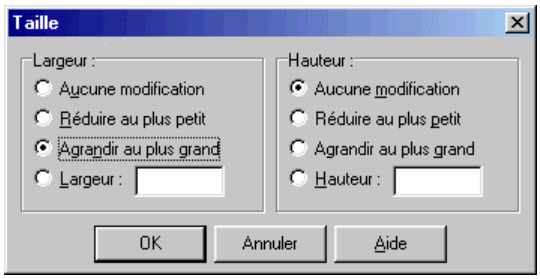

Figure 5. La gestion de la taille de plusieurs composants

Si vous le souhaitez, vous pouvez consulter l'aide pour comprendre la signification des options d'alignement (Cliquez sur « *Aide* » ou appuyez sur F1). Choisissez une option d'alignement horizontal et une option d'alignement vertical puis validez la boîte de dialogue.

- Vous pouvez aussi faire apparaître une palette d'alignement avec la commande « *Voir / Palette d'alignement* ». Cette palette restera présente à l'écran et vous permettra d'envoyer des commandes d'alignement aux composants sélectionnés :

Figure 6. La palette d'alignement

Cliquez sur la palette d'alignement, puis appuyez sur la touche F1 afin de charger le système d'aide et d'étudier le rôle de chacune de ces icônes. Notez aussi que lorsque la souris stationne sur une icône un court message explicatif de la commande apparaît dans une bulle d'aide jaune.

Pratique

Déposez plusieurs composants sur une fiche et testez tous ces outils d'alignement.

Pratique

Concevez l'aspect graphique de la fenêtre suivante :

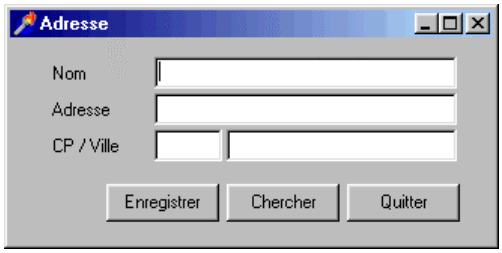

Figure 7. Exemple de fenêtre à concevoir

Cette fenêtre comprends des composants de la palette standard de composants : 3 boutons OK, 3 labels A et 4 zones de saisie ab|.

1.2 UN PREMIER PROGRAMME AVEC DELPHI

1.2.1 Le programme a créer

Démarrez un nouveau projet sous *Delphi*, placez deux boutons dans la fenêtre principale du programme et sauvegardez le projet de manière à obtenir quelque chose ressemblant à ceci :

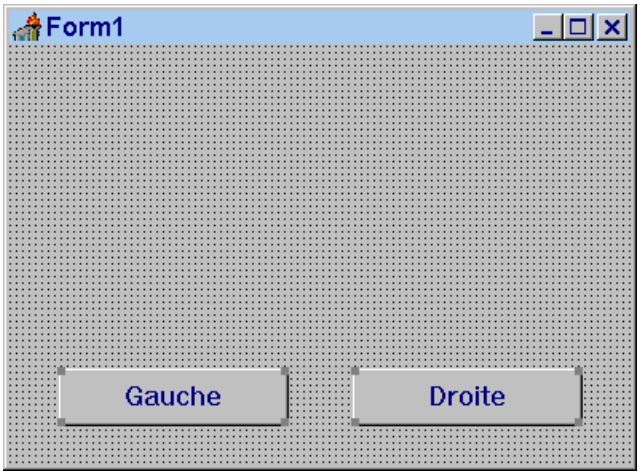

Figure 8. Projet de fenêtre avec deux boutons

Nous désirons faire les choses suivantes avec cette fenêtre :

- Lorsque nous cliquons sur le bouton de gauche :
  - Le bouton doit afficher « GAUCHE » en majuscules (et nous devons nous assurer que l'autre bouton affiche « Droite » en minuscule).
  - La fenêtre doit passer en bleu.
- Lorsque nous cliquons sur le bouton de droite :
  - Le bouton doit afficher « DROITE » en majuscules (et nous devons nous assurer que l'autre bouton affiche « Gauche » en minuscule).
  - La fenêtre doit passer en vert.

Le nom des composants

Nous vous rappelons l'habitude suivante des programmeurs sous Delphi (Voir le cours « *Delphi partie 1 — Le Pascal Objet* ») :

> **Une habitude de programmation veut sous Delphi que l'on nomme ses objets en mettant un préfixe qui rappelle la classe à laquelle appartient l'objet**. Cette abréviation est souvent constituée de la première lettre de la classe suivie des consonnes du nom de la classe (En éliminant les consonnes en double). Par exemple « *FntrClient* » pour une fenêtre client, « *FchrFacture* » pour un fichier de factures.

Aussi, après avoir placé les boutons gauche et droit, utilisez l'inspecteur d'objet, et modifiez la propriété « *Name* » des boutons afin qu'ils s'appellent « `BtnGauche` » et « `BtnDroite` ».

1.2.2 Le code généré par Delphi

Après avoir conçu la fenêtre précédente, examinons le code se trouvant dans le source associé à cette fenêtre :

```
unit Fenetre2;

interface

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

type
  TForm1 = class(TForm)
    BtnGauche: TButton;
    BtnDroite: TButton;
  private
    { Private-déclarations }
  public
    { Public-déclarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

Delphi a conçu pour nous une unité pour stocker le code lié à cette fenêtre.

La partie « `interface` » contient plusieurs choses :

- Une déclaration « `uses` » établissant les liens vers toutes les unités nécessaires au fonctionnement de la fiche qui a été conçue.
- Une section « `type` » dans laquelle est déclarée une classe permettant de gérer des fenêtres telles que celle que nous avons conçue.
- Une section « `var` » dans laquelle est déclarée une instance de cette classe.

La partie « `implementation` » ne contient aucun code si ce n'est une sorte de commentaire « `{$R *.DFM}` ».

Les directives de compilation

Les *directives de compilation* sont des ordres particuliers envoyés au compilateur ou à l'éditeur de lien. Elles servent à paramétrer le fonctionnement de ces outils.

Les *directives de compilation* ne font pas partie du langage (ce ne sont pas des déclarations ni des instructions), mais sont destinées à régler d'autres aspects de la génération du programme. La plupart de ces options de compilation sont aussi accessibles par les onglets « *Compilateur* » et « *Lieur* » de la boîte de dialogue « *Projet / Options* ».

Elles peuvent aussi figurer dans le code sous forme de « pseudo commentaire » avec la syntaxe : « `{$...}` ».

« `{$R *.DFM}` » est une directive de compilation qui demande d'intégrer au programme, sous un format particulier, les paramètres initiaux de la fiche (Le contenu du fichier « *Fenetre2.dfm* »).

La classe générée

Delphi a généré automatiquement une classe pour gérer les fiches de la même nature que celle que nous avons conçue graphiquement.

Ceci signifie, entre autres, que si nous le souhaitons nous pourrons créer plusieurs instances de cette fiche.

Le nom de la classe reprend la propriété « *Name* » de la fiche préfixé de la lettre « *T* »[11].

Cette classe hérite de la classe « *TForm* » de la *VCL*. « *TForm* » est la super classe de toutes les fenêtres, et boîtes de dialogue dont a besoin un programmeur sous *Windows*. Cette classe est déclarée dans l'unité « *Forms* » de la *VCL*.

Grâce à l'héritage et au polymorphisme, le programmeur peut personnaliser la sous-classe comme il le souhaite : y introduire de nouveaux états et en modifier ou compléter les comportements.

Comme nous avons ajouté deux boutons à la fenêtre, *Delphi* a créé deux variables d'instance, « `BtnGauche` » et « `BtnDroite` » permettant de manipuler ces boutons :

> **Tout composant inséré dans une fiche en mode conception mène à la création d'une variable d'instance dont le nom correspond à la propriété « `Name` » du composant.**

La fiche « auto créée »

Par défaut, *Delphi* crée automatiquement une instance de chaque nouvelle fiche crée à partir du concepteur de fiche.

Pour permettre de manipuler cette fiche par défaut, *Delphi* a créé une section « `var` » avec une variable portant le même nom que la fiche.

Les fiches auto créées seront automatiquement détruites lors de la terminaison du programme. Ceci respecte un principe implicite de la *Programmation Orientée Objet* : celui qui créée un objet est responsable du fait que cet objet soir ultérieurement détruit.

[11] En toute rigueur, avec les habitudes des programmeurs Delphi, nous aurions du changer le nom de la fiche en « `FrmFormation` » et la classe se serait appelée « `TFrmFormation` ».

Où le programmeur peut-il intervenir dans le code ?

En théorie, le programmeur peut intervenir n'importe où dans le source conçu par *Delphi* et ajouter, modifier ou supprimer le code qu'il souhaite.

Cependant :

- Le code compris entre le nom de la classe de la fiche (« `TForm1` » dans notre exemple) et le premier mot clef « `private` » est géré automatiquement par *Delphi* (Intervenir dans cette section demande donc quelques précautions).
- *Delphi* complète automatiquement la première déclaration « `uses` » en fonction des composants ajoutés à la fiche.

En conclusion :

N'effectuez aucune modification manuelle entre le nom de la classe de la fiche et entre la déclaration « `private` » (A moins de savoir ce que vous faites réellement).

Ne supprimez pas la première déclaration « uses », et si vous y supprimez des unités qui ne sont plus nécessaires assurez-vous de laisser celles dont la classe a besoin.

1.2.3 Le programme principal généré par Delphi

Ouvrez le source du programme principal et examinons son contenu :

```
program Projet02;

uses
  Forms,
  Fenetre2 in 'FENETRE2.PAS' {Form1};

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Le programme principal comprend les éléments suivants :

- Une déclaration « `uses` » vers toutes les unités nécessaires. Notez le commentaire en face de l'unité correspondant à une fiche — Delphi analyse automatiquement ces commentaires afin de gérer diverses boîtes de dialogue.
- Une directive de compilation « `{$R *.RES}` » lie le fichier « Projet02.res » au programme généré. Il s'agit d'un fichier contenant des paramètres liés au programme tels que l'icône représentant ce programme.
- Les instructions du programme principal manipulent un objet « *Application* » de la classe « *TApplication* ».

L'objet « Application »

Un pur *Programme Orienté Objet* est organisé à partir d'un objet principal qui encapsule le fonctionnement global de l'application.

Cet objet réalise toutes les initialisations nécessaires, crée tous les objets et fournit des mécanismes permettant de gérer le fonctionnement global du programme.

Lorsque le programme se termine cet objet est détruit. Il procède alors à tout le travail de nettoyage nécessaire, ceci incluant la destruction des objets appartenant à l'application.

Dans le cadre d'un programme *Delphi*, c'est objet s'appelle « *Application* » et est de la classe « *TApplication* ». Il réalisera toutes les initialisations nécessaires dans la *VCL*.

Delphi génère dans le programme principal le code nécessaire à l'instanciation des fiches auto créées, grâce à la méthode « *CreateForm* », puis appelle une méthode « *Run* » de l'objet *Application* afin de lancer l'exécution réelle du programme.

Où le programmeur peut-il intervenir dans le code ?

En théorie le programmeur peut intervenir à n'importe quel endroit du code. Cependant :

- Les lignes « *CreateForm* » et « *Run* » sont gérées automatiquement par *Delphi*.
- La liste des « uses » est complétée automatiquement par *Delphi*.

En conclusion

Le programmeur devra être prudent lorsqu'il intervient dans la liste des « uses » (et devra éviter de le faire à moins qu'il ne sache réellement ce qu'il fait).

Le programmeur devra intervenir avant le premier « Application.CreateForm » ou après le « Application.Run ».

1.2.4 La modification du comportement

Nous allons maintenant modifier le *comportement* de ces objets pour que :

- Un clic sur le bouton *Gauche* change la couleur de la fenêtre en bleu et met le titre du bouton en majuscules.
- Un clic sur le bouton *Droit* change la couleur de la fenêtre en vert et met le titre du bouton en majuscules.

Pour ceci, commencez par faire un double clic sur le bouton *Gauche* : L'éditeur de texte apparaît, le source du programme a été modifié pour que vous puissiez saisir le code d'une méthode `TForm1.BtnGaucheClick`.

De la même manière, lorsque vous aurez saisi le code adéquat, faite un double clic sur le bouton *Droit* ; l'éditeur de texte apparaîtra pour que vous saisissiez le code d'une méthode `TForm1.BtnDroiteClick`.

Voici le code (en gras) à introduire pour obtenir l'effet souhaité :

```
unit Fenetre2;

interface

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

type
  TForm1 = class(TForm)
    BtnGauche: TButton;
    BtnDroite: TButton;
    procedure BtnGaucheClick(Sender: TObject);
    procedure BtnDroiteClick(Sender: TObject);
  private
    { Private-déclarations }
  public
    { Public-déclarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.BtnGaucheClick(Sender: TObject);
begin
  Color := clBlue;
  BtnGauche.Caption := 'GAUCHE';
  BtnDroite.Caption := 'Droite';
end;

procedure TForm1.BtnDroiteClick(Sender: TObject);
begin
  Color := clGreen;
  BtnGauche.Caption := 'Gauche';
  BtnDroite.Caption := 'DROITE';
end;

end.
```

Lorsque vous avez double cliqué sur le bouton gauche, *Delphi* a créé automatiquement une méthode « `BtnGaucheClick` ». Cette méthode sera appelée automatiquement à chaque fois que l'utilisateur clique sur ce bouton.

Notez que lorsque le programme appelle la méthode « `BtnGaucheClick` » il reçoit un paramètre « `Sender` » de la classe « *TObject* ». Ce paramètre permet d'identifier l'objet qui sollicite le traitement de la méthode « `BtnGaucheClick` », en l'occurrence dans le programme que nous avons écrit « `Sender` » identifiera toujours le bouton gauche, « `BtnGauche` » (Autrement dit « `Sender = BtnGauche` »).

Dans le code de cette méthode, « `Color := clBlue ;` » modifie la couleur de fond de la fiche pour la passer en bleu. En effet la méthode « `BtnGaucheClick` » est une méthode de la classe « `TForm1` », et « `Color` » identifie une propriété d'une fiche.

Remarque

Un grand nombre de composant de la VCL possède une propriété « `Color` » qui permet d'en modifier la couleur de fond.

Une couleur peut être déterminée par une constante identifiant une couleur ou par une valeur identifiant le pourcentage de vert de rouge et de bleu que nous affectons à cette couleur.

Les constantes d'identification de couleur commencent toutes par le préfixe « `cl...` ». Elles sont de deux natures :

- Une constante de couleur peut identifier une couleur réelle, par exemple « `clBlue` » pour le bleu, « `clGreen` » pour le vert…
- Une constante de couleur peut identifier une couleur système. Par exemple « `clBtnFace` » identifie la couleur de la face des boutons standard de *Windows*.

Pour avoir un résumé rapide des codes de couleurs, vous pouvez examinez la propriété « `Color` » de la fiche dans l'inspecteur d'objet. Une liste permet de choisir un de ces codes de couleurs.

Remarque

La manière de choisir une valeur pour une propriété dans l'inspecteur d'objet peut changer en fonction du type ou de la classe de la propriété.

Un code de couleur peut être choisi dans la liste proposée.

Un double clic sur un code de couleur fait apparaître une boîte de dialogue permettant de choisir une couleur personnalisée grâce à une boîte de dialogue standard de Windows — Delphi rapatriera le code adéquat dans l'inspecteur d'objet :

Figure 9. La boîte de dialogue de choix de couleur

Lorsque vous manipulez la fiche avec l'inspecteur d'objet, vous retrouvez cette propriété « `Color` », et vous devez remarquer qu'elle a été initialisée avec la valeur « `clBtnFace` ». Si vous modifiez cette propriété avec l'inspecteur d'objet, vous verriez la fiche changer de couleur. Nous sommes ici en train de faire la même chose avec l'instruction « `Color := clBlue ;` ».

L'instruction « `BtnGauche.Caption := 'GAUCHE';` » manipule la variable d'instance « `BtnGauche` » de la fiche. Cette variable identifie un objet de la classe « `TButton` » qui possède une propriété « `Caption` ».

Remarque

Un grand nombre de composants de la *VCL* possède une propriété « `Caption` » de type « `string` ».

Cette propriété identifie le texte principal du composant. Pour une fenêtre, ce sera le contenu de la barre de titre, pour un bouton le texte affiché sur ce bouton...

L'instruction « `BtnGauche.Caption := 'GAUCHE';` » force donc le texte « *GAUCHE* » dans le bouton de gauche.

De même l'instruction « `BtnDroite.Caption := 'Droite';` » force le texte « *Droite* » dans le bouton de droite.

Donc lorsqu'on clique sur le bouton de gauche :

- On change la couleur de la fiche en bleu.
- On force le texte du bouton de gauche en majuscule.
- On force le texte du bouton de droite en minuscule.

Un clic sur le bouton de droite se gère avec un code semblable qui peut se comprendre avec les explications précédentes.

Pratique

Réalisez l'exemple précédent et vérifiez qu'il fonctionne comme il se doit.

2. GESTION D'EVENEMENT VERSUS POLYMORPHISME

Il existe deux grandes méthodes permettant de personnaliser le comportement d'un composant de la VCL :

- L'écriture d'un gestionnaire d'événement adéquat comme l'a montré l'exemple du chapitre précédent.
- L'utilisation du polymorphisme en surchargeant une méthode virtuelle de la super classe afin de l'adapter aux besoins de la classe courante.

2.1 LE POLYMORPHISME ET LA SURCHARGE DE METHODE

Cette deuxième technique est une technique qui sera fréquemment utilisée par un concepteur de composant. Elle pourra aussi être utilisée par un programmeur d'application qui intégrera des mécanismes objets après avoir classifié les fiches et les boîtes de dialogue dont il a besoin.

Nous allons voir ici un exemple simple de cette technique. Supposons que nous désirions modifier l'apparence de la fiche de l'exemple précédent, non pas en y ajoutant un composant mais en modifiant le dessin apparaissant sur le fond de la fiche :

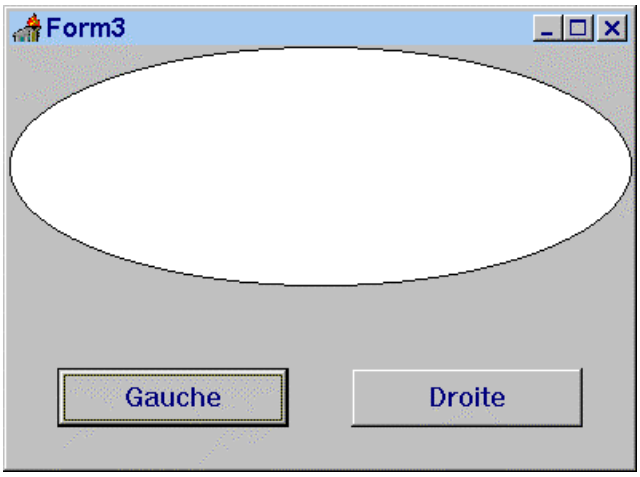

Figure 10. Exemple 3 : tracer une ellipse dans la fenêtre

Plusieurs solutions sont à notre disposition. La solution que nous avons retenue pour illustrer le polymorphisme repose sur les principes suivants :

- Chaque contrôle possède une méthode virtuelle protégée « `Paint` ». A chaque fois qu'un contrôle a besoin d'être dessiné, *Windows* le fait savoir à l'application ; la *VCL* récupère cette demande du système et appelle la méthode « `Paint` » du contrôle correspondant.
- La plupart des contrôles de la *VCL* possèdent une propriété « `Canvas` » de la classe « `TCanvas` ». Un canevas permet de dessiner sur quelque chose (une fiche, un composant, une imprimante…). Le canevas d'un composant permet de dessiner sur ce composant, il possède les méthodes nécessaires à la réalisation d'un dessin.

Nous allons réaliser le dessin du fond d'écran avec le code suivant :

```
unit Fenetre3;

interface

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

type
  TForm1 = class(TForm)
    BoutonGauche: TButton;
    BoutonDroite: TButton;
    procedure BoutonGaucheClick(Sender: TObject);
    procedure BoutonDroiteClick(Sender: TObject);
  private
    { Private-déclarations }
  protected
    procedure Paint; override;
  public
    { Public-déclarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{ le code des méthodes BoutonGaucheClick et BoutonDroiteClick }
{ est identique aux extraits de listing précédents            }

procedure TForm1.Paint;
begin
  inherited Paint;
  Canvas.Ellipse (0, 0, 390, 150);
end;

end.
```

L'essentiel des explications concernant ce listing a été vu précédemment. La seule instruction méritant quelques explications complémentaires est « `Canvas.Ellipse (0, 0, 390, 150);` ».

Le canevas possède une méthode « `Ellipse` » recevant 4 paramètres. Cette méthode trace une ellipse sur le composant en respectant le rectangle d'encombrement passé en paramètre (X1, Y1, X2 et Y2 déterminent les positions du coin supérieur gauche et du coin inférieur droit de ce rectangle d'encombrement.).

Pratique

Réalisez cet exemple.

Sachant qu'un canevas possède aussi une méthode « `TextOut` » possédant 3 paramètres (Une position X, une position Y est un texte à afficher), affichez aussi un message au cœur de l'ellipse.

2.2 LA DELEGATION D'EVENEMENTS

Une des définitions d'*événement* est : « *Tout ce qui arrive, tout fait qui intervient dans une continuité et en caractérise un moment…* » (Dictionnaire Logos de Bordas).

On pourrait résumer un événement à « *Il se passe quelque chose* ».

Le quelque chose en question peut avoir plusieurs sources :

- Le système signale quelque chose qui demande un traitement. Par exemple une fenêtre a besoin d'être redessinée sur l'écran (par exemple par ce qu'il y avait quelque chose qui la masquait et ce quelque chose a disparu), autre exemple : un timer lancé par l'application est arrivé à échéance, ou encore : l'utilisateur a cliqué sur un bouton…
- Un composant désire prévenir un autre composant de son changement d'état.

2.2.1 La gestion de messages sous Windows

On pourrait comparer Windows à un moteur de génération de messages. Chaque message envoyé à l'application prévient celle-ci d'un changement d'état détecté dans le système. De la même manière une application peut envoyer des messages vers le système (qui peuvent à leur tour être envoyés à l'application ou à d'autres applications).

Un message est une structure de données qui permet de coder un événement et de fournir des paramètres permettant de le décrire.

Le schéma ci-dessous décrit comment les composants de la *VCL* et *Windows* travaillent ensemble afin de répondre aux besoins du programme :

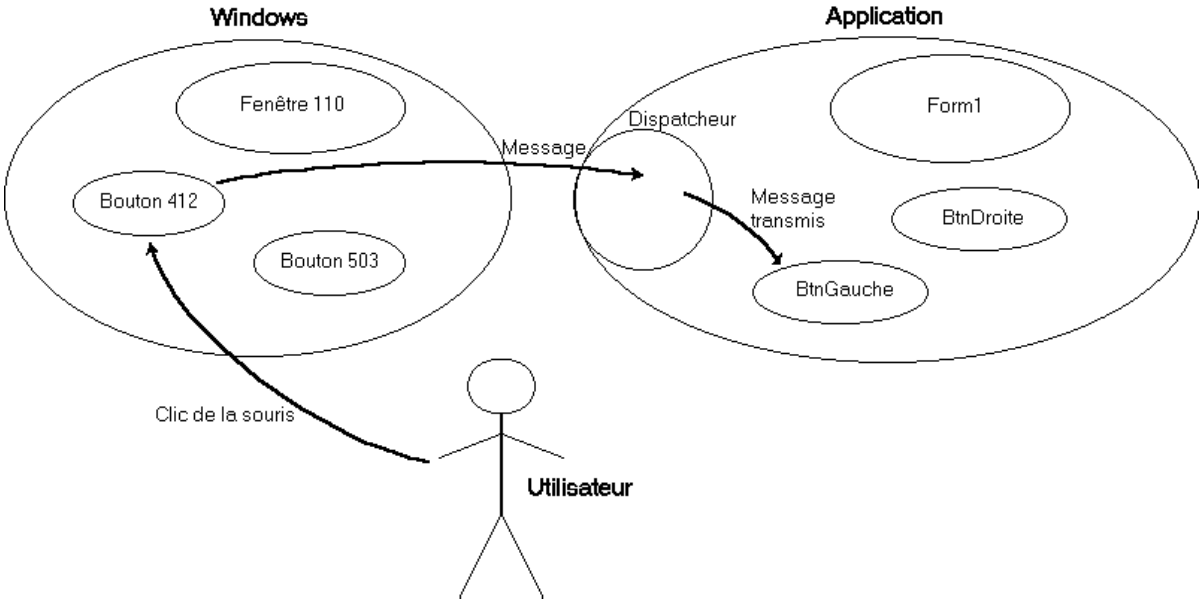

Figure 11. La gestion de messages de Windows

Ce schéma illustre les éléments suivants :

- *Windows* pilote plusieurs de structure de données[12] interne afin de gérer l'affichage de base d'une fenêtre et de 2 boutons : une fenêtre portant le numéro 110, un bouton portant le numéro 412 et un autre portant le numéro 503[13].
- Dans l'application, la *VCL* a créé 3 objets permettant de superviser ces 3 structures gérées par *Windows*. C'est comme cela que la *VCL* décharge le programmeur de toute la complexité liée à *Windows* et à la manière dont une application interagit avec le système.
- Lorsqu'un événement intervient au niveau du système (par exemple l'utilisateur a cliqué avec la souris sur l'écran) :
  - *Windows* détecté l'élément concerné.
  - *Windows* prévient l'application de l'événement et sur quel élément porte cet événement (en précisant le numéro de l'événement). Pour ceci, *Windows* appelle une fonction particulière dans l'application.
- La fonction de l'application recevant les messages de *Windows* contient un mécanisme d'analyse qui permet de retrouver le composant concerné dans l'application.
- Si le composant possède une méthode pour traiter cet événement, le mécanisme d'analyse appelle automatiquement cette méthode.

L'étude de ce mécanisme permet de comprendre comment l'exemple de la section précédente fonctionne :

- *Windows* détecte que la fenêtre a besoin d'être redessinée.
- *Windows* prévient l'application avec le message « *WM_PAINT*[14] » pour la fenêtre 110.
- Le dispatcheur retrouve le composant « *Form1* » qui supervise la fenêtre 110.
- La méthode « *Form1.Paint* » est appelée automatiquement.
- Comme nous avons surchargé cette méthode, le programme trace une ellipse.

Vérification de la compréhension

Qu'est-ce qu'un message au sens *Windows* du terme ?

A quoi servent les composants de la *VCL* qui ont été instanciés dans une application ?

Comment un message système est-il transformé en l'appel d'une méthode sur un composant ?

[12] Il s'agit bien de structure de données. *Windows* est écrit de manière structurée et non orientée objets.

[13] Dans la terminologie *Windows*, ces numéros d'identification sont appelés des « *témoins* » ou en anglais des « *handles* ». Ceci explique la propriété en lecture seule « *Handle* » présente dans la plupart des composants de la *VCL*.

[14] Constante définie au sein de Windows permettant d'identifier les messages signifiant « *cet élément a besoin d'être dessiné* ».

2.2.2 La délégation d'événement

Le mécanisme de gestion de messages que nous venons d'examiner permet déjà d'écrire des applications qui fonctionneraient avec *Windows*.

Il offre cependant un inconvénient énorme. Considérons par exemple une application qui contiendrait environ 500 boutons répartis entre plusieurs fenêtres :

> Il faudrait sous classer 500 fois la classe « *TButton* » et à chaque fois surcharger la méthode adéquate afin de gérer le clic sur ces boutons. Pour ajouter à la complexité de la chose, chaque bouton devrait analyser la fenêtre à laquelle il appartient afin d'agir sur celle-ci.
>
> Nous aurions alors environ 500 sous classe de la classe « TButton » et quelque complexité dans leur écriture.
>
> Hormis la complexité que cela amènerait au programmeur, l'atelier de développement graphique de *Delphi* s'en trouverait alourdi aussi.

Le mécanisme de la *délégation d'événement* résout élégamment cette difficulté :

- Tous les composants de la *VCL* susceptible de recevoir des événements possèdent des propriétés de style « *gestionnaire d'événement* ».
- Une propriété de style « *gestionnaire d'événement* » est un pointeur sur une méthode d'un autre objet, méthode qui sera appelée si l'événement se produit.
- Ce mécanisme, écrit dans les composants de la *VCL*, permet de rediriger l'événement à l'endroit où il doit être traité.
- Pour que le récepteur de l'événement puisse identifier, s'il le souhaite, l'objet qui lui a délégué l'événement, les gestionnaires d'événement possèdent en général un paramètre « `Sender : TObject` ».

Le schéma suivant illustre ceci :

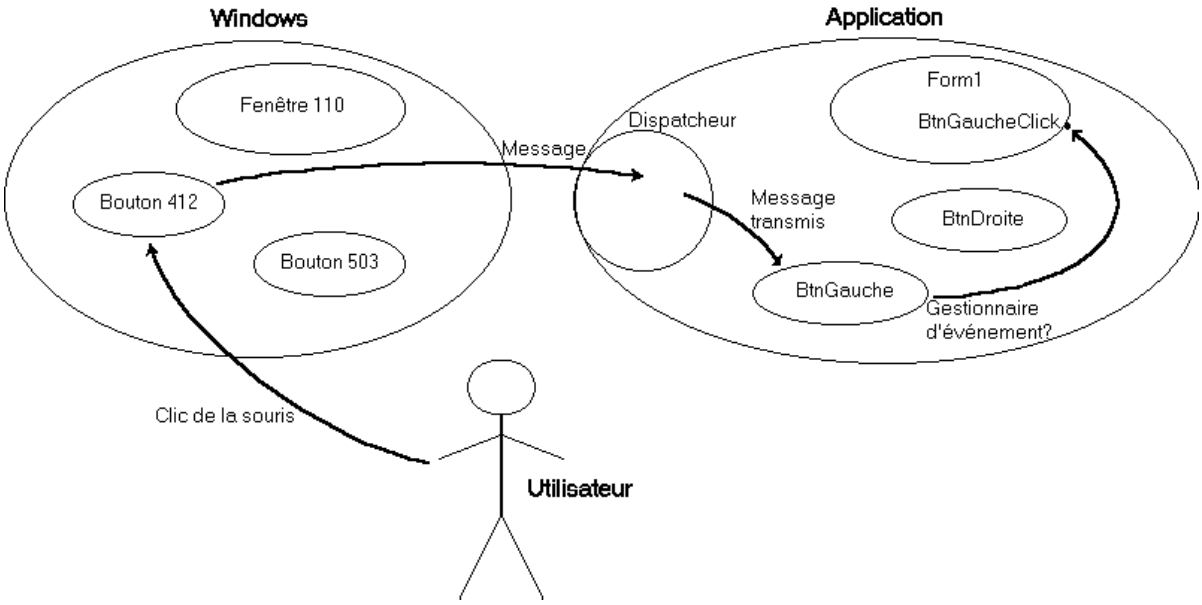

Figure 12. La délégation d'événement

Exemple

Nous allons reprendre l'exemple précédent, mais allons écrire un seul gestionnaire d'événement pour les deux boutons.

Pour ceci :

- Supprimez le code de la méthode « `BtnDroiteClick` » pour qu'il devienne :

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

- Enregistrez le source. Lors de l'enregistrement de ce source, *Delphi* procède à une analyse de celui-ci. Comme il se rend compte que le corps de la méthode du gestionnaire d'événement est vide, il retire la méthode et sa déclaration dans la classe[15].
- Inspectez le composant « `BtnDroite` ».
- Cliquez sur l'onglet « *événement* » de l'inspecteur d'objet (Vous y découvrez, entre autres, qu'un bouton est capable de générer d'autres événements que le clic) :

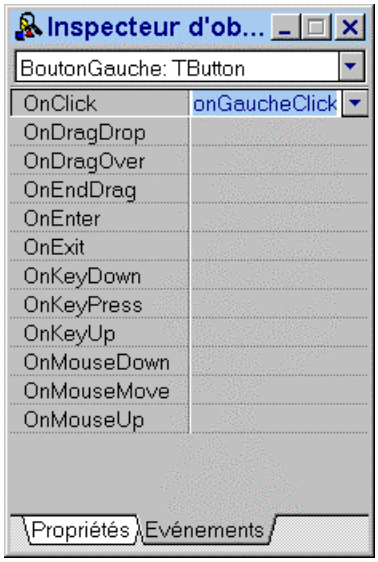

Figure 13. L'inspecteur d'objet, les événements

- Grâce à la liste déroulante en face de l'événement « `OnClick` », associez l'événement « `BtnGaucheClick` » au bouton « `BtnDroite` ».

[15] Nous vous rappelons que cette méthode a été déclarée avant le mot clef « `private` » et que cette portion de code est maintenue automatiquement par *Delphi*.

- Corrigez le code de la méthode « `BtnGaucheClick` » pour qu'il devienne :

```
procedure TForm1.BtnGaucheClick(Sender: TObject);
begin
   if Sender = BtnGauche then
      begin
      Color := clBlue;
      BtnGauche.Caption := 'GAUCHE';
      BtnDroite.Caption := 'Droite';
      End
   else
      begin
      Color := clGreen;
      BtnGauche.Caption := 'Gauche';
      BtnDroite.Caption := 'DROITE';
      End;
end;
```

Remarque

Cette technique ne vous est présentée qu'à titre d'exemple et que pour illustrer deux manières de programmer différentes. Ceci ne signifie pas que cette technique est meilleure ou moins bonne que la précédente. Il faudra que vous jugiez en fonction des circonstances et de ce que vous souhaitez faire pour choisir entre une technique ou l'autre.

Dans le cadre des exemples évoqués notre choix va en fait à la première technique.

Pratique

Réalisez l'exemple précédent.

Exemple

Avec la *VCL*, nous pouvons facilement réaliser des messages d'aide contextuelle sous la forme de bulle jaune.

Aussi désirons-nous faire les choses suivantes :

- Lorsque la souris s'arrête sur le bouton gauche, une bulle jaune doit apparaître signalant « *Change la couleur de la fiche en bleu* ».
- Lorsque la souris s'arrête sur le bouton de droite, une bulle jaune doit apparaître signalant « *Change la couleur de la fiche en vert* ».
- Une barre d'état en bas de la fenêtre répercutera aussi ce message d'aide contextuelle.

Les deux premières étapes sont faciles à faire :

- La plupart des contrôles, entre autres les « `TButton` » possèdent une propriété « `Hint`[16] » et une propriété « `ShowHint`[17] ».
- Basculez[18] la propriété « `ShowHint` » à « `true` » pour chacun de ces composants, et inscrivez le texte de l'aide dans la propriété « `Hint` ».

Exécutez le programme pour vérifier le fonctionnement du mécanisme d'affichage des bulles d'aide.

La troisième étape va être un peu plus délicate. En effet, l'affichage d'un texte d'aide dans une zone non standard de la VCL se fait en interceptant un événement de l'objet « *Application* »… Malheureusement, cet objet ne peut être inspecté[19], il va nous falloir installer le gestionnaire d'événement avec du code :

- Commencez par mettre une barre d'état au pied de la fiche :
  - Sélectionnez l'onglet « Win32 » de la palette de composant.
  - Cliquez sur le composant « `StatusBar` » de la palette de composant (4ième composant à partir de la droite).
  - Cliquez sur la fiche, une barre d'état est installée en bas de la fiche.
- Pour écrire le gestionnaire d'événement affichant le texte d'aide :
  - Le système d'aide de *Delphi* nous permet de découvrir la syntaxe de l'événement « `TApplication.OnHint` » : « `procedure (Sender : TObjet) of object` » ; c'est la syntaxe d'un pointeur sur une procédure d'objet recevant un paramètre « `Sender` ».
  - En conséquence de quoi, nous écrivons les instructions suivantes dans la classe « `TForm1` » :

```
{...}
type
  TForm1 = class(TForm)
    {...}
  private
    { Déclarations privées}
    procedure Conseil (Sender : TObject);
    {...}
  end;
{...}
implementation
{...}
procedure TForm1.Conseil(Sender: TObject);
begin
  StatusBar1.SimpleText := GetLongHint(Application.Hint);
end;
end.
```

[16] A traduire par « *Truc et astuce* » ou « *Conseil* ».

[17] A traduire par « *Montrer le truc et l'astuce* » ou « *Montrer le conseil* ».

[18] Lorsqu'une propriété est de type ordinal, et qu'une liste déroulante permet le choix d'une valeur pour cette propriété dans l'inspecteur d'objet, vous pouvez double cliquer sur la valeur sélectionner pour passer à la valeur suivante.

[19] L'objet « *Application* » est inaccessible en mode conception.

« `GetLongHint` » est une fonction de la *VCL* permettant de mettre en forme le conseil à afficher.

« `Application.Hint` » est le conseil que l'application doit afficher.

« `SimpleText` » est une propriété d'une barre d'état permettant d'afficher un texte (dans le cadre d'une barre d'état simple ne possédant pas plusieurs zones d'affichage).

- Nous devons maintenant installer ce gestionnaire d'événement :
  - Nous allons installer ce gestionnaire d'événement lors de la création de la fenêtre dans *Windows*. Pour ceci, inspectez les événements de la fenêtre et double cliquez sur l'événement « `OnCreate` ».
  - Delphi installe un gestionnaire d'événement dans lequel nous pouvons programmer l'instruction suivante :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := Conseil;
end;
```

« `Application.OnHint` » est la propriété correspondant à l'événement « `OnHint` » de l'application.

« `Conseil` » est la méthode que nous venons d'écrire pour gérer cet événement.

Pratique

Réalisez le programme précédent.

2.2.3 La différence entre le constructeur et l'événement « OnCreate »

Les figures « Figure 11. La gestion de messages de Windows » et « Figure 12. La délégation d'événement » permettent de comprendre le rapport entre les objets de la *VCL* instanciés par l'application *Delphi* et les structures de données gérées par *Windows*.

Un grand nombre d'objets de la *VCL* est destiné à gérer pour le programmeur des composants *Windows* (identifiés dans *Windows* par un « *Témoin* » — un « *Handle* »).

Les constructeurs et les destructeurs instancient et détruisent les objets de la *VCL*. Mais lors de l'appel du constructeur, l'élément correspondant n'est pas encore créé dans *Windows*.

L'événement « `OnCreate` » est déclenché depuis *Windows* après avoir reçu l'ordre de créer le composant et après l'avoir créé. A ce stade le composant existe aussi bien en tant que structure de *Windows* qu'en tant qu'objet de l'application.

De la même manière l'événement « `OnDestroy` » est déclenché lors de la destruction de la structure dans *Windows*, et le destructeur correspond à la destruction de l'objet.

Les conséquences de ceci sont les suivantes :

- Si vous avez besoin de manipuler l'objet en interaction avec Windows, vous devrez travailler avec les événements « `OnCreate` » et « `OnDestroy` ».
- Si vous n'avez besoin de ne manipuler que des éléments ayant trait à l'objet (hors contexte Windows), vous pouvez travailler indifféremment avec les événements « `OnCreate` » et « `OnDestroy` » ou avec le constructeur et le destructeur.
- Si vous voulez faire des travaux en amont de Windows, travaillez avec le constructeur et le destructeur.

Vérification de la compréhension

Pourquoi, dans la plupart des cas, pourrez-vous travailler avec les événements « `OnCreate` » et « `OnDestroy` » lorsque vous aurez des intialisations et des travaux de nettoyage à faire en rapport avec le cycle de vie d'une fenêtre.

3. LA PRISE EN MAIN DE LA BIBLIOTHEQUE

3.1 LE CANEVAS

Tous les *composants* graphiques contiennent un objet « `Canvas` » permettant de dessiner dans la fenêtre ou le *composant* grâce à ses *propriétés* et ses *méthodes*.

Voici les principales *propriétés* du *Canevas* permettant de paramétrer le dessin :

- **PEN** (de la classe **TPEN**) : caractéristique du « *crayon* » utilisé pour tracer les lignes ou le contour des formes (couleur, épaisseur,...)
- **BRUSH** (de la classe **TBRUSH**) : caractéristique du « *pinceau* » utilisé pour colorier l'intérieur des formes (couleur, hachures,...).
- **FONT** (de la classe **TFONT**) : caractéristique de la police utilisée pour écrire du texte (nom de la police, taille, couleur...).

Voici les principales *méthodes* permettant de dessiner dans une fenêtre ou un *composant* :

- **MOVETO** : déplace le *crayon* sans tracer de trait pour commencer le tracé d'une ligne.
- **LINETO** : trace un trait depuis la dernière position du *crayon*.
- **RECTANGLE** (et **ROUNDRECT**) : trace un rectangle — Les dessins fermés utilisent les caractéristiques de **PEN** pour leur contour et celle de **BRUSH** pour leur intérieur.
- **ELLIPSE** : trace un cercle ou une ellipse.
- **TEXTOUT** : Affiche du texte (chaîne de caractères).

D'autres *méthodes* existent par exemple pour tracer un arc de cercle, un camembert,... (Consultez l'aide sur la rubrique **TCANVAS**), et manipuler tout ce qui a à voir avec l'affichage dans une fenêtre ou un composant.

Pratique

En interceptant l'*événement* **ONMOUSEDOWN** d'une fenêtre (interception d'un clic sur un bouton de la souris), faites l'opération suivante :

- Si c'est un clic du bouton droit, affichez les coordonnées de la souris où le clic a eu lieu (ces coordonnées **X** et **Y** sont des paramètres de la méthode **TFORM1.FORMMOUSEDOWN**).

Pour savoir quel est le bouton qui a été activé, testez le paramètre **BUTTON** de type **TMOUSEBUTTON** (valeur **MBRIGHT**).

Pratique

Tracer tous les déplacements de la souris, lorsque le bouton gauche de la souris est enfoncé (**Button = mbLeft**). Pour ceci :

- Créez un nouveau *champ* logique, **DessinEnCours**, dans la *classe* **TForm1** (*champ privé*) pour mémoriser l'état du bouton gauche de la souris.
- Interceptez les *événements* **OnMouseDown**, **OnMouseUp** et **OnMouseMove** pour gérer ce tracé :
  - Sur **OnMouseDown** : déplacer le *crayon* aux coordonnées de la souris (**LineTo**), et faire monter le champ logique **DessinEnCours** à **true**.
  - Sur **OnMouseMove** : si **DessinEnCours** est vrai : dessiner (**MoveTo**).
  - Sur **OnMouseUp** : Faire descendre le champ **DessinEnCours** à **false**.

Pratique

Modifiez les *propriétés* de la police (**Font**) et testez l'effet des diverses *propriétés* (police, taille, couleur,...) — Consultez l'aide sur **TFont** si nécessaire).

Modifiez les *propriétés* du *crayon* (**Pen**) et testez l'effet des diverses *propriétés* (couleur, taille du crayon,...) — Consultez l'aide sur **TPen** si nécessaire).

3.2 MEMORISER LE DESSIN

Dans l'exemple précédent, remarquez que lorsque que vous changez la taille de la fenêtre, vous pouvez perdre une partie du dessin — il faut donc mémoriser tous les points et le reconstituer en surchargeant la méthode « `Paint` » de la fenêtre ou en gérant son événement « `OnPaint` ».

3.2.1 La classe TList

Parmi les objets à notre disposition dans la VCL, nous trouvons la classe « `TList` ». Les objets de la classe « `TList` » permettent de gérer facilement des tableaux dynamiques de pointeurs.

Aussi allons nous procéder de la manière suivante :

- Dans une unité à part, nous gérerons une nouvelle classe permettant de mémoriser et dessiner une ligne constituée d'une série de point.
- Nous allons créer une nouvelle classe, « `TCoordonnees` » permettant de mémoriser sous forme objet des coordonnées x et y.
- Dans la classe permettant de gérer une liste de points, nous allons intégrer un objet « `TList` » pour stocker plusieurs points.
- Nous allons munir cette classe d'une méthode permettant d'ajouter facilement un point, de dessiner le tableau de points sur un canevas[20] et une dernière méthode pour réinitialiser le tableau de point.

[20] Cette méthode pourra ultérieurement être utilisée pour imprimer le dessin.

Commençons par étudier la classe « `TList` », vous pouvez examiner l'aide sur cette classe afin d'étudier les propriétés et les méthodes de celle-ci. Voici un résumé des éléments qui nous intéressent :

- La propriété en lecture seule « `Count` » permet de déterminer combien d'éléments ont été stockés dans la liste.
- La propriété « `Items […]` » permet de retrouver un pointer d'indice donné dans la liste (L'indice doit être compris entre 0 et « Count-1 »).
- La méthode « `Add(…)` » permet d'ajouter un pointeur dans la liste.
- La méthode « `Clear` » permet de réinitialiser le tableau de pointeur de manière à ce que la liste soit vide.

Remarque

La propriété « `Items […]` » est une propriété de type tableau ; c'est aussi une propriété de type « *tableau par défaut* ». Ceci signifie que lorsque nous manipulons cette propriété en l'appliquant à un objet, nous ne sommes pas obligés d'en préciser le nom.

Par exemple :

```
{…}
var
   Liste : TList;
   P : pointer;
Begin
{…}
P := Liste.Items [i];
{ ou }
P := Liste[i];
{…}
end.
```

Les instructions « `Liste.Items [i]` » et « `Liste[i]` » sont équivalentes.

Remarque

Seules les propriétés de type tableau peuvent être déclarées comme étant des propriétés par défaut.

Dans le cadre de la classe « `TList` » cela a déjà été fait dans la *VCL*.

Remarque

Les objets « `TList` » sont des tableaux de pointeurs, nous pourrions donc y stocker autre chose que des objets. Un objet « `TList` » ne peut donc prendre en charge la destruction des objets qui y sont stockés, il nous incombe donc d'écrire le code de destruction.

Ces explications préliminaires étant faites, nous pouvons réaliser les classes nécessaires.

- Créez une nouvelle unité (« *Fichier / Nouveau… / Nouveau / Unité* ».
- Enregistrez cette unité sous le nom « *Graphiques.pas* ».
- Créez le code suivant dans cette unité.

```
unit Graphiques;

interface

uses classes, graphics;

type
  TCoordonnees = class
    x : integer;
    y : integer;
    constructor Create (x0, y0 : integer);
    end;

  TDessin = class
    protected
      Liste : TList;
    public
      constructor Create;
      destructor Destroy; override;
      procedure AjoutePoint (x, y : integer);
      procedure Clear;
      procedure Dessiner (Canvas : TCanvas);
      end;

implementation

{ TCoordonnees }

constructor TCoordonnees.Create(x0, y0: integer);
begin
  inherited Create;
  x := x0;
  y := y0;
end;

{ TDessin }

procedure TDessin.AjoutePoint(x, y: integer);
begin
  Liste.Add (TCoordonnees.Create(x, y));
end;

procedure TDessin.Clear;
  var
    i : integer;
begin
  for i := Liste.Count-1 downto 0 do
    TCoordonnees (Liste[i]).Free;
  Liste.Clear;
end;

constructor TDessin.Create;
begin
  inherited Create;
  Liste := TList.Create;
end;

procedure TDessin.Dessiner(Canvas: TCanvas);
  var
    i : integer;
    n : integer;
begin
  n := Liste.Count-1;
  if n <= 0 then
    exit;
  with TCoordonnees (Liste [0]) do
    Canvas.MoveTo (x, y);
  for i := 1 to n do
    with TCoordonnees (Liste [i]) do
      Canvas.LineTo (x, y);
end;

destructor TDessin.Destroy;
begin
  Clear;
  Liste.Free;
  inherited Destroy;
end;

end.
```

3.2.2 La mémorisation du dessin

Nous pouvons alors installer le code suivant dans la fiche :

```
unit fTexte;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, Graphiques;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure FormPaint(Sender: TObject);
  private
    { Déclarations privées}
    DessinEnCours : boolean;
    Dessin : TDessin;
  public
    { Déclarations publiques}
  end;
var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  DessinEnCours := false;
  Dessin := TDessin.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  Dessin.Free;
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Invalidate;
  DessinEnCours := true;
  Dessin.Clear;
  Dessin.AjoutePoint(x, y);
  Canvas.MoveTo (x, y);
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Dessin.AjoutePoint(x, y);
  Canvas.LineTo (x, y);
  DessinEnCours := false;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if DessinEnCours then
    begin
    Dessin.AjoutePoint(x, y);
    Canvas.LineTo (x, y);
    end;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  Dessin.Dessiner(Canvas);
end;
end.
```

Remarque

Notez comment le cycle de vie des objets a été géré :

- Lorsque la fiche est créée dans *Windows*, l'événement « `OnCreate` » crée l'objet local « `Dessin` ».
- Lors de la construction de l'objet « `TDessin` », l'objet « `Liste` » est créé.
- Lorsque la fiche est détruite dans *Windows*, l'événement « `OnDestroy` » détruit l'objet local « `Dessin` ».
- Lors de la destruction du dessin, la méthode « `Clear` » détruit toutes les coordonnées qui ont été stockées dans le dessin, puis l'objet « `Liste` » est détruit.

Pratique

Réalisez l'application précédente qui servira de support aux exercices pratiques de ce chapitre.

3.3 LES MENUS

Deux composant de Delphi permettent de gérer des menus :

- « `TMainMenu` » gérant un menu de fiche placé juste sous la barre de titre de la fenêtre.
- « `TPopupMenu` » gérant un menu contextuel pouvant être associé à un composant.

3.3.1 Un menu principal

Pour ajouter un *menu* :

- Prenez un objet *MainMenu* dans la *palette de composants* avec la souris, et déposez le dans la fiche qui doit être gérée par ce *menu*.
- Faites un double clic sur ce *menu* pour faire apparaître le *concepteur de menu*. (Ou un clic du bouton droit et choix du *concepteur de menu* dans le menu contextuel).
- Dans le *concepteur de menu*, vous pouvez faire un clic du bouton droit pour charger un modèle de menu (insérer depuis un modèle) — Nous allons charger ainsi le « *menu Fichier* ».
- Pour compléter le menu, assurez-vous que l'*inspecteur d'objets* soit positionné sur la rubrique « *Caption* » (titre), puis cliquez sur une case vide du menu (à droite du *menu* ou en bas du *sous-menu*) pour créer un nouvel item de menu — Entrez son libellé, validez avec la touche ↵, le *concepteur de menu* passe automatiquement à l'élément suivant (les touches Inser et Suppr permettent d'insérer ou de supprimer des éléments de menus). Le « *et commercial* », « `&` », peut être utilisé dans un élément de *menu* pour souligner le caractère qui le suit — celui-ci servira alors de *raccourci* de clavier (en combinaison avec la touche Alt). Utilisez l'*inspecteur d'objet* pour paramétrer complètement l'élément de *menu* (chargez l'aide depuis l'*inspecteur d'objet* si nécessaire).

Pratique

Concevez un menu qui ait l'allure suivante :

```
&Fichier                          &Options                  &A propos
&Nouveau                          &Couleur du trait...
&Ouvrir...                        &Epaisseur du trait...
&Enregistrer                      Couleur de &Fond..
En&registrer sous...
&Imprimer
&Configuration d'impression...
&Quitter
```

Lorsque ceci est fait, exécutez le programme — le *menu* est présent, mais rien ne se passe lorsque vous choisissez un élément dans le *menu*.

Remarque

Inspectez le fiche et consultez la propriété « `Menu` ». Remarquez que cette propriété a été automatiquement liée au menu que vous venez d'installer.

Grâce à cette propriété, vous pouvez changer dynamiquement le menu d'une fiche à l'exécution. Il vous suffit de créer plusieurs « `TMainMenu` » puis d'affecter un de ces menus à la propriété « `Menu` ».

Pour obtenir une action, il faut installer un gestionnaire d'*événement* pour chaque item de menu le nécessitant. Pour créer un gestionnaire d'événement, vous pouvez double cliquer sur un item de menu depuis le concepteur de menu (ou inspecter les événements de cet item et créer le menu depuis l'inspecteur d'objet). Vous pouvez aussi opérer depuis la fiche, le menu est présent au sommet de la fiche en mode conception, cliquez sur l'item que vous voulez programmer.

Créez par exemple le gestionnaire d'événement suivant pour la commande « *Fichier / Nouveau* » :

```
procedure TForm1.Nouveau1Click(Sender: TObject);
begin
   ShowMessage ('J''ai cliqué sur Fichier nouveau');
end;
```

La procédure « `ShowMessage` » appelle directement une fonction dans Windows permettant d'afficher le message passé en paramètre.

Créez aussi le gestionnaire d'événement suivant pour la commande « *Fichier / Quitter* » :

```
procedure TForm1.Quitter1Click(Sender: TObject);
begin
   Close;
end;
```

La méthode « `Close` » d'une fiche ou d'une boîte de dialogue demande la fermeture de celle-ci.

3.3.2 Un menu surgissant

Pour ajouter un *menu surgissant* :

- Prenez un objet *PopupMain* dans la *palette de composants* avec la souris, et déposez le dans la fiche contenant le composant qui doit être gérée par ce *menu*.
- Faites un double clic sur ce *menu* pour faire apparaître le *concepteur de menu*. (Ou un clic du bouton droit et choix du *concepteur de menu* dans le menu contextuel).
- Dans le *concepteur de menu* vous pouvez procéder comme pour un menu *MainMenu*.
- Inspectez le ou les composants auxquels vous souhaitez lier ce menu et inspectez la propriété « PopupMenu » ; dans la liste des menus disponibles sélectionnez le menu que vous désirez attacher à ce ou ces composants.

Pratique

Concevez un menu qui ait l'allure suivante :

```
&Nouveau
&Ouvrir...
&Enregistrer
En&registrer sous...
-
&Imprimer
&Configuration d'impression...
-
&Couleur du trait...
&Epaisseur du trait...
Couleur de &Fond..
-
&Quitter
```

Attachez les événements « `Nouveau1Click` » et « `Quitter1Click` » aux items « `&Nouveau` » et « `&Quitter` » du menu surgissant.

Remarque

Vous pouvez travailler par couper coller entre les deux menus. Pour cela :

- Ouvrez le premier menu dans le concepteur de menu.
- Sélectionnez les items « `&Nouveau` » à « `&Quitter` » (Cliquez sur « `&Nouveau` », appuyez sur la touche ⇧ Shift et appuyez sur « `&Quitter` »).
- Appuyez sur les touches Ctrl+C.
- Ouvrez le deuxième menu.
- Appuyez sur les touches Ctrl+V.

3.3.3 La programmation de raccourcis de clavier

Les items de menus possèdent une propriété « `ShortCut` ». Choisissez éventuellement un raccourci dans la liste des raccourcis disponible. A l'exécution, lorsque l'utilisateur utilisera cette combinaison de touches la commande associée au menu sera exécutée. (Voyez aussi l'aide sur cette propriété « `ShortCut` » pour voir toutes les manières de la paramétrer).

3.4 LA GESTION DE BOITE DE DIALOGUE

Dans cette section nous allons voir différents cas simples de gestion de boîte de dialogue.

3.4.1 La boîte de dialogue « A propos »

- Nous allons ici charger un modèle de boîte de dialogue « A Propos » et le modifier. Pour ajouter une boîte de dialogue à partir d'un modèle :
- Dans le menu de *Delphi*, choisissez « *Fichier / Nouveau…* ».
- Sélectionnez l'onglet « *Fiche* » :

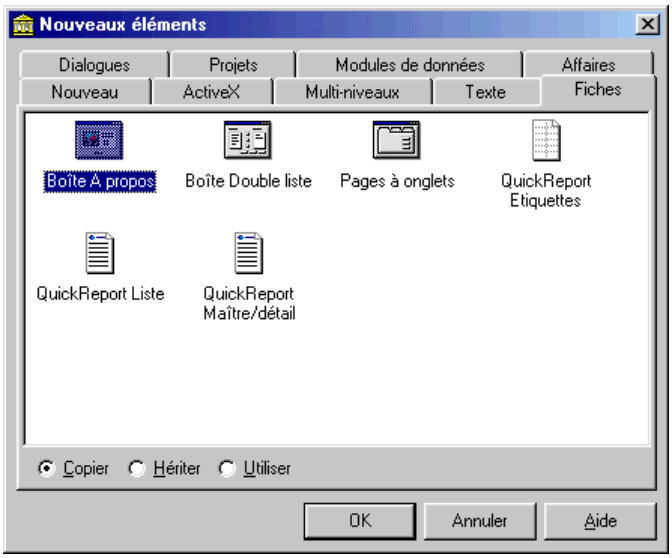

Figure 14. L'ajout d'une nouvelle boîte de dialogue à un projet

- Choisissez le modèle « *A Propos* » et validez la boîte de dialogue.
- Modifiez le contenu de cette fenêtre à votre guise grâce à l'inspecteur d'objet et grâce à la palette de composants.
- Donnez un nom cohérent à la boîte de dialogue (Propriété « `Name` ») et enregistrez-la.

Remarque

Dans la boîte de dialogue ci-dessus, vous remarquez trois boutons radio :

- « *Copier* » : le modèle de fiche sélectionné est importé dans le projet.
- « *Hériter* » : Nous aurons l'occasion de revenir sur cette option qui permet de classifier les fenêtres et les boîtes de dialogue au sein d'un même projet ou entre plusieurs projets.
- « *Utiliser* » : Le projet utiliserait alors la boîte de dialogue stockée dans le référentiel de Delphi.

Dans le cadre de l'utilisation d'un modèle la meilleure utilisation est « *Copier* », ceci romps le lien avec le référentiel de Delphi.

Pratique

Créer une fenêtre « *A Propos* », appelez-la « `FrmAPropos` » et stockez-la dans l'unité « `fAPropos` ». Examinez le code généré par *Delphi* dans la nouvelle unité pour gérer cette boîte de dialogue.

Puis gérez l'événement du bouton « &`A Propos` » du menu de l'application pour faire surgir cette fiche. Pour ceci utilisez la méthode « `ShowModal` » et appliquez-la à la boîte de dialogue.

Cette méthode donne la main à la boîte de dialogue et aucun autre composant de l'application ne peut avoir la main tant que l'exécution de la boîte de dialogue n'est pas terminée.

```
Implemntation

   uses fAPropos3;                  { déclare l'unité où est définie }
                                    { la boîte de dialogue }
   {...}
   procedure TForm1.APropos1Click(Sender: TObject);
   begin
      FrmAPropos.ShowModal;
   end;
```

Remarque

Si vous avez omis d'ajouter la déclaration « `uses fAPropos3;` », *Delphi* non seulement le détectera à la compilation, mais il sera aussi capable d'analyser la nature de l'erreur et vous proposera d'ajouter automatiquement cette déclaration.

Validez la proposition de Delphi et relancez la compilation.

3.4.2 La boîte de dialogue de changement de couleur

Windows possède une fonction permettant d'afficher une boîte de dialogue de sélection de couleur — fonction de paramétrage complexe. *Delphi* encapsule cette boîte de dialogue dans la classe « `TcolorDialog` ».

La propriété « `Color` » de la boîte de dialogue permet d'initialiser la couleur dans la boîte de dialogue, puis de lire la couleur qui a été choisie par l'utilisateur.

La méthode « `Execute` » fait apparaître la boîte de dialogue (de façon modale) et donne la main à l'utilisateur. Si celui-ci choisit une couleur, la méthode renvoie « `true` » (récupérez alors la couleur dans « `Color` ») ; sinon elle renvoie « `false` ».

Pour installez une boîte de dialogue de changement de couleur :

- Sélectionnez l'onglet « *Dialogues* » de la palette de composants.
- Cliquez sur l'objet « *ColorDialog* ».
- Cliquez sur la fiche.
- Inspectez ce nouveau composant et donnez un nom adéquat dans la propriété « `Name` » (par exemple « `ClrdlgTrait` »).

Dans le gestionnaire d'événement adéquat, gérons maintenant cette boîte de dialogue avec un code tel que :

```
procedure TForm1.Couleurdutrait1Click(Sender: TObject);
begin
  // ClrdlgTrait.Color := ?
  if ClrdlgTrait.Execute then
    begin
    // gérer le changement de couleur du trait
    end;
end;
```

L'intégration de la couleur au dessin

Pour pouvoir terminer cet exemple, il nous faut revoir la classe de gestion de dessin pour intégrer la notion de couleur.

```
unit Graphiques;

interface

uses classes, graphics;

type
  {...}
  TDessin = class
    protected
      Liste : TList;
    public
      Couleur : TColor;
      constructor Create (Coul : TColor);
      destructor Destroy; override;
      procedure AjoutePoint (x, y : integer);
      procedure Clear;
      procedure Dessiner (Canvas : TCanvas);
      end;

implementation

{...}

constructor TDessin.Create (Coul : TColor);
begin
  inherited Create;
  Liste := TList.Create;
  Couleur := Coul;
end;

procedure TDessin.Dessiner(Canvas: TCanvas);
  var
    i : integer;
    n : integer;
begin
  Canvas.Pen.Color := Couleur;
  n := Liste.Count-1;
  if n <= 0 then
    exit;
  with TCoordonnees (Liste [0]) do
    Canvas.MoveTo (x, y);
  for i := 1 to n do
    with TCoordonnees (Liste [i]) do
      Canvas.LineTo (x, y);
end;

end.
```

Nous pouvons maintenant intégrer la gestion de la couleur au niveau de la fiche :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DessinEnCours := false;
  Dessin := TDessin.Create (Canvas.Pen.Color);
end;

procedure TForm1.Couleurdutrait1Click(Sender: TObject);
begin
  ClrdlgTrait.Color := Canvas.Pen.Color;
  if ClrdlgTrait.Execute then
    begin
    Dessin.Couleur := ClrdlgTrait.Color;
    invalidate;
    end;
end;
```

Pratique

Réalisez l'exemple précédent.

Pensez à attacher cet événement aussi bien au menu principal qu'au menu surgissant.

Pratique

Utilisez la même boîte de dialogue pour modifier la couleur de fond de la fiche.

Pour ceci, gérez l 'événement suivant :

```
procedure TForm1.CopuleurdeFond1Click(Sender: TObject);
begin
//???
end;
```

Dans cet événement :

- Initialisez la couleur de la boîte de dialogue avec la couleur de la fiche.
- Appelez la méthode « `Execute` » de cette boîte de dialogue.
- Si l'utilisateur a validé une nouvelle couleur, affectez-la à la fiche.

3.4.3 La création d'une boîte de dialogue simple

Nous allons ici construire une *boîte de dialogue* pour demander l'épaisseur du trait :

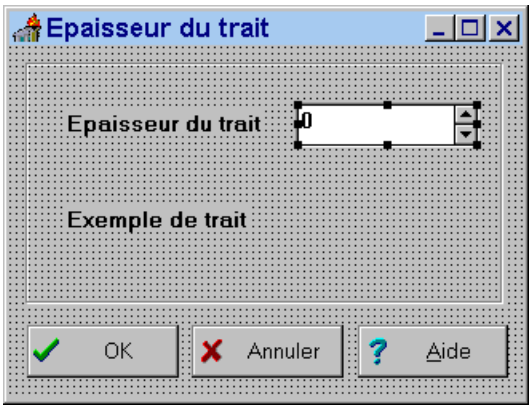

Figure 15. La boîte de dialogue "épaisseur du trait"

Pour concevoir cette boîte de dialogue, procédez comme vous l'avez fait pour la boîte de dialogue « *A Propos* », en choisissant le modèle « *Dialogue standard*[21] », puis :

- Dans la *palette de composants*, choisissez un *composant* de la *classe* « *Label* » placez-le et modifiez en le contenu pour qu'il affiche « *Epaisseur du trait* ».
- Placez de la même manière un autre « *Label* » avec le contenu « *Exemple de trait* » (sous ce texte nous dessinerons un trait de la taille donnée.
- Sous l'onglet « *Exemples* » de la palette de composants, prenez un « *SpinEdit*[22] » et placez-le à droite de l'épaisseur du trait. Modifiez ses propriétés comme suit :
  - « `MinValue` » à 1 ; la valeur minimum possible dans ce champ.
  - « `MaxValue` » à 100 ; la valeur maximum possible dans ce champ.
- Si la boîte de dialogue que vous avez chargée possède un bouton « *Aide* », supprimez-le, ce bouton n'est pas géré par notre application.
- Donnez des noms adéquat au composants (par exemple « `FrmEpaisseur` » pour la boîte de dialogue, « `SpnedtEpaisseur` » pour le contrôle de saisie numérique, « `BtnOK` » et « `BtnAnnuler` » pour les boutons) et enregistrez la fiche sous un nom adéquat (par exemple « `fEpaisseur` »).

[21] Cette boîte de « *dialogue standard* » se trouve sous l'onglet « *Dialogues* » de la commande « *Fichier / Nouveau…* »

[22] Un contrôle « `TSpinEdit` » est dédié à la saisie de valeurs numériques. Il ressemble à un contrôle « `TEdit` » et possède deux flèches sur le bord droit qui permettent d'augmenter ou diminuer la valeur numérique avec la souris.

Pratique

Créez cette boîte de dialogue.

Créez une méthode qui gère l'événement « *Option / Epaisseur du trait* » du menu. Avec un code ressemblant à celui-ci :

```
procedure TForm1.Epaisseurdutrait1Click(Sender: TObject);
begin
  FrmEpaisseur.SpnedtEpaisseur.Value := Canvas.Pen.Width;
  if FrmEpaisseur.ShowModal = mrOk then
    begin
    Canvas.Pen.Width := FrmEpaisseur.SpnedtEpaisseur.Value;
    end;
end;
```

Remarque

La méthode « `ShowModal` » retourne une valeur permettant d'identifier comment on est sorti de la boîte de dialogue. (Sortie sur un bouton, sortie en fermant la fenêtre, etc...)

En inspectant les boutons « `BtnOK` » et « `BtnCancel` », vous découvrez une propriété « `ModalResult` » qui vaut « `mrOK` » ou « `mrCancel` ». Ces propriétés des boutons déterminent la valeur de retour qu'aura « `ShowModal` » (Consultez éventuellement l'aide de *Delphi* pour cette *propriété* « `ModalResult` ».

La propriété « `Value` » d'un « `SpinEdit` » permet d'initialiser ou récupérer la valeur entière du composant.

Pensez à attacher la commande « *Epaisseur du trait* » du menu surgissant au même gestionnaire d'événement.

Exécutez le programme, et notez que son fonctionnement est encore imparfait, la nouvelle épaisseur de trait n'est prise en compte que pour les prochains dessins.

Pratique

Ajoutez une nouvelle propriété « Epaisseur » à la classe « `TDessin` » de l'unité « `Graphiques` » et gérez-la comme nous avons géré la couleur :

- Déclarez une variable d'instance « `Epaisseur : integer ;` » comme nous avons déclaré la variable « `Couleur` ».
- Modifiez le constructeur afin qu'il reçoive une couleur et une épaisseur, initialisez la couleur en conséquence.
- Revoyez la méthode de dessin afin d'initialiser la propriété « `Canvas.Pen.Width` » avec l'épaisseur.

Revoyez le gestionnaire d'événement de modification d'épaisseur dans l'esprit du gestionnaire d'événement de modification d'épaisseur.

Pratique

Revenez dans le source de la boîte de dialogue et créez une méthode privée « `ExempleTrait` » qui dessine l'exemple de trait à l'endroit adéquat.

```
procedure TFrmEpaisseur.ExempleTrait;
begin
  with Canvas do begin
    Pen.Width := 1;
    Rectangle (40, 120, 290, 136);
    Pen.Width := SpnedtEpaisseur.Value; { il faudrait tester si   }
    MoveTo ( 80, 128);                  { le trait n'est pas plus }
    LineTo (250, 128);                  { large que le rectangle  }
    end;

end;
```

Créez une méthode qui intercepte le dessin de la fiche (événement « `OnPaint` »), et un autre qui intercepte tous les changements de valeur dans le « `SpinEdit` » (événement « `OnChange` »). Dans ces gestionnaires d'événement appelez la méthode précédente.

3.5 L'IMPRESSION

3.5.1 La configuration de l'imprimante

De la même manière que nous avions une boîte de dialogue préfabriquée dans *Windows* pour gérer le choix de couleur , « *TColorDialog* », nous en avons une autre, « *TPrinterSetupDialog*[23] », pour gérer le choix de l'imprimante.

Pratique

Déposez un objet « *TPrinterSetupDialog* » dans la fiche depuis l'onglet « Dialogue » de la palette de composants.

Interceptez l'*événement* géré par le menu « *Fichier / Configuration de l'impression...* » du menu de notre application et exécutez cette *boîte de dialogue* en lui appliquant la méthode « `Execute` ».

Associez l'item « *Configuration de l'impression...* » du menu surgissant au même gestionnaire d'événement.

[23] Cette boîte de dialogue permet de gérer le mécanisme d'impression de Delphi qui est utilisé dans la section suivante.

3.5.2 L'impression

L'impression dans une application *Delphi* est une opération simple. En effet la *VCL* définit un objet « `Printer` » (imprimante) ; cette imprimante contient une propriété « `Canvas` » complètement compatible avec le canevas qui nous a permis de dessiner le contenu de la fenêtre.

Les travaux d'impressions sont aussi simple que :

```
uses {...} Printers ;
procedure TForm1.Imprimer1Click(Sender: TObject);
begin
with Printer do
  begin
  BeginDoc;                          { début de l'impression }
  Dessin.Dessiner (Canvas);          { Printer.Canvas est disponible }
  EndDoc;                            { fin de l'impression }
  end;
end;
```

Remarque

Voyez aussi dans l'aide la *méthode* « `NewPage` » pour gérer les changements de page, voyez aussi les autres *méthodes*.

Avant de lancer l'impression on peut souhaiter faire apparaître une boîte de dialogue demandant la confirmation de l'impression avec une validation du choix de l'imprimante. Ceci peut être réalisé facilement avec un objet « `TPrintDialog` » de l'onglet « *Dialogues* » de la palette de composants. Déposez un tel objet dans la fiche et modifiez le gestionnaire d'événement de la manière suivante :

```
procedure TForm1.Imprimer1Click(Sender: TObject);
begin
  if PrintDialog1.Execute then
    with Printer do
      begin
      BeginDoc;                          { début de l'impression }
      Dessin.Dessiner (Canvas);          { Printer.Canvas est disponible }
      EndDoc;                            { fin de l'impression }
      end;
end;
```

Pratique

Intégrez le code d'impression du dessin.

3.6 L'AJOUT D'UNE BARRE D'OUTILS

Nous allons maintenant ajouter une barre d'outils à notre application :

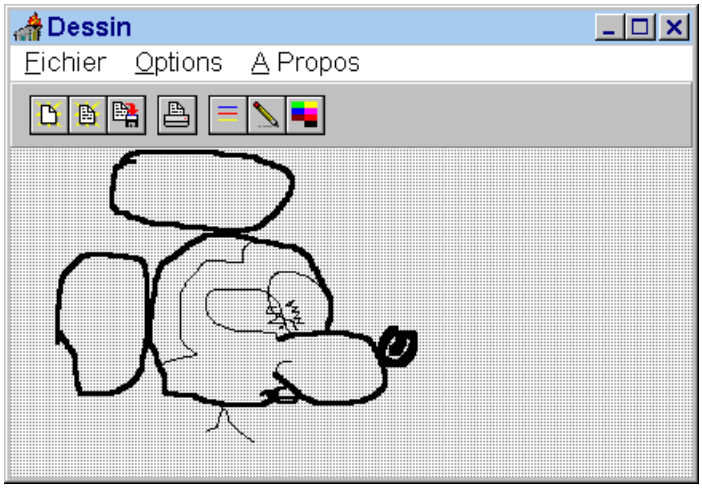

Figure 16. L'ajout d'une barre d'outils

Il s'agit d'une opération relativement aisée. Nous ajoutons dans la fenêtre un composant « `TPanel` » (*Panneau*) grâce à la palette de composants. Celui-ci comme la plupart des composants possède une propriété « `Align` » qui peut être utilisée pour forcer la position et l'alignement d'un composant avec le composant qui le contient. Forcez la valeur de cette propriété à « `alTop` » (Alignement en haut). Le panneau va se fixer en haut du composant qui le contient et y restera toujours.

Il ne reste plus qu'à habiller ce composant :

- Mettez une chaîne vide dans la propriété « `Caption` ».
- Placer dans ce panneau des « `TSpeedButton` » (de l'onglet « *Suppléments* » de la palette de composants).
- Habillez chacun d'eux avec une icône (*BitMap* ou *Icône*) grâce à leur propriété « `Glyph` » (le répertoire « *Program Files\Fichiers commun\Borland shared\Images\Button* » comprend une collection d'images prête à être intégrées dans des *SpeedButton* — copiez les icônes désirées dans votre répertoire de travail — réalisez les dessins manquants grâce à Paint Brush).
- Attachez un gestionnaire d'événement à chacun de ces *SpeedButton* (double clic sur le bouton).

Remarque

Notez que les *composants* placés dans le panneau sont intégrés à celui-ci. Ainsi si vous changez « `Panel1.Align` » en « `alBottom` » (Alignement en bas), la *barre d'outils* se déplace en bas avec ses boutons.

Mise en pratique

Réalisez la barre d'outils avec les boutons ci-dessus.

3.7 LA GESTION DE FICHIER

3.7.1 Les objets de gestion de fichier

Les fonctions de gestion de fichier du *Pascal* sont toujours disponibles, cependant dans un langage orienté objets, nous trouvons aussi une gestion de fichier orientée objets.

Nous trouvons une classe abstraite, « `TStream` » base de tous les objets fichiers. En fait un *stream* est un *flux de données*, c'est à dire quelque chose dans lequel on peut lire ou écrire des données — Ceci n'est pas restreint à une gestion de fichier, mais peut comprendre aussi un stockage en mémoire EMS, l'émission de données sur une liaison série...

Une classe dérivée de « `TStream` », « `TFileStream` », permet de lire et d'écrire dans un fichier. Les *méthodes* principales concernant l'accès à un *flux* sont :

- « `TFileStream.Create (Nom de fichier, mode d'ouverture)` » : construction de l'objet et ouverture du fichier. (Le mode d'ouverture peut valoir les valeurs suivantes (ou une somme de ces valeurs) : « `fmOpenRead` », « `fmOpenWrite` », « `fmOpenReadWrite` », « `fmCreate` » ou d'autres valeurs telle que « `fmShare` » pour gérer le partage de fichiers.
- « `TStream.Read (Buffer, Taille du buffer)` » : Lecture dans le buffer (de type variable) du nombre d'octets spécifiés.
- « `TStream.Write (Buffer, Taille du buffer)` » : Ecriture dans le *flux* depuis le buffer (de type variable) du nombre d'octets spécifiés.
- D'autres méthodes et propriétés permettent de déplacer le *pointeur de fichier* dans le *flux*.

3.7.2 Gestion de fichier objet

La technique de travail que nous allons utiliser consiste à munir chaque objet de méthodes pour enregistrer et lire un objet dans ou depuis un flux. Donc nous allons créer pour chaque classe à enregistrer constructeur et une méthode pour gérer la lecture et l'enregistrement :

```
unit Graphiques;

interface

uses classes, graphics;

type
  TCoordonnees = class
    x : integer;
    y : integer;
    constructor Create (x0, y0 : integer); overload;
    constructor Create (Flux : TStream); overload;
    procedure Write  (Flux : TStream);
    end;
```

```
  TDessin = class
    protected
      Liste : TList;
    public
      Couleur : TColor;
      Epaisseur : integer;
      constructor Create (Coul : TColor; Ep : integer); overload;
      constructor Create (Flux : TStream); overload;
      constructor Create (Fichier : string); overload;
      destructor Destroy; override;
      procedure Write  (Flux : TStream); overload;
      procedure Write  (Fichier : string); overload;
      procedure AjoutePoint (x, y : integer);
      procedure Clear;
      procedure Dessiner (Canvas : TCanvas);
      end;

implementation

uses SysUtils;

{ TCoordonnees }

constructor TCoordonnees.Create(Flux: TStream);
begin
  inherited Create;
  Flux.Read (x, sizeof (x));
  Flux.Read (y, sizeof (y));
end;

procedure TCoordonnees.Write(Flux: TStream);
begin
  Flux.Write (x, sizeof (x));
  Flux.Write (y, sizeof (y));
end;

{ TDessin }

constructor TDessin.Create(Flux: TStream);
  var
    i : integer;
    n : integer;
begin
  inherited Create;
  Liste := TList.Create;
  Flux.Read (Couleur,   sizeof (Couleur));
  Flux.Read (Epaisseur, sizeof (Epaisseur));
  Flux.Read (n,         sizeof (n));
  n := n-1;
  for i := 0 to n do
    Liste.Add (TCoordonnees.Create (Flux));
end;

constructor TDessin.Create(Fichier: string);
  var
    F : TFileStream;
begin
  F := TFileStream.Create (Fichier, fmOpenRead);
  try
    Create (F);
  finally
    F.Free;
  end;
end;

procedure TDessin.Write(Flux: TStream);
  var
    i : integer;
    n : integer;
begin
  Flux.Write (Couleur,   sizeof (Couleur));
  Flux.Write (Epaisseur, sizeof (Epaisseur));
  Flux.Write (Liste.Count, sizeof (Liste.Count));
  n := Liste.Count-1;
  for i := 0 to n do
    with TCoordonnees (Liste [i]) do
      Write  (Flux);
end;
```

```
procedure TDessin.Write(Fichier: string);
  var
    F : TFileStream;
begin
  F := TFileStream.Create (Fichier, fmCreate);
  try
    Write (F);
  finally
    F.Free;
  end;
end;

end.
```

Les objets de la classe de dessin sont maintenant prêts pour être enregistrés ou pour être créés à partir d'un fichier.

3.7.3 Les boîtes de dialogue de gestion de fichier

Windows intègre une fonction qui permet d'afficher des boîtes de dialogue de gestion de fichier (ouverture, sauvegarde,...). Ces fonctions de paramétrage complexe ont été encapsulées dans la *VCL* par les classes « `TOpenDialog` » et « `TSaveDialog` ». Celles-ci s'utilisent un peu comme la boîte de dialogue de gestion des couleurs. Prenons par exemple l'enregistrement de fichier :

- Cliquez sur un composant « `TSaveDialog` » de l'onglet « Dialogues » de la palette de composants ?
- Cliquez sur la fiche.
- Inspectez l'objet « `SaveDialog1` » :
  - Modifiez sa propriété « `Name` » en « `SvdlgDessin` ».
  - Modifiez sa propriété « `Title` » en « `Enregistrement de dessin` ».

    Il s'agit du titre qui sera affiché au sommet de la boîte de dialogue.
  - Modifiez sa propriété « `DefaultExt` » en « `PTS` ».

    Il s'agit de l'extension par défaut. Si le fichier sélectionné ne porte pas d'extension, le composant ajoutera automatiquement cette extension au fichier sélectionné.
  - Modifiez sa propriété « `Filter` » en « `Dessin de points|*.PTS` » et « `Tous les fichiers|*.*` ».

    La *propriété* « `Filter` » permet de préciser une série de noms et de masques de fichiers. Ceux-ci servent à gérer une liste *combo* en bas à gauche de la boîte de dialogue pour sélectionner des fichiers d'un type donné.

    Dans l'inspecteur d'objet, cliquez sur les points de suspension à droite de la valeur de la propriété pour faire apparaître une boîte de dialogue permettant définir ces types ; introduisez deux lignes avec les caractéristiques ci-dessus.
  - Vous pourrez initialiser ou récupérer le nom du fichier dans la propriété « `FileName` ».

Il ne reste plus qu'à ajouter un gestionnaire d'événement à la commande « Fichier / Enregistrer sous… » qui ressemble à ceci :

```
procedure TForm1.Enregistrersous1Click(Sender: TObject);
begin
  SvdlgDessin.FileName := NomFichier;
  if SvdlgDessin.Execute then
    begin
    NomFichier := SvdlgDessin.FileName;
    Dessin.Write (NomFichier);
    end;
end;
```

De la même manière, nous pouvons déposer une boîte de dialogue « `TOpenDialog` » avec le nom « `OpndlgDessin` » et écrire un gestionnaire d'événement tel que le suivant pour la commande « `Fichier / Ouvrir` » :

```
procedure TForm1.Ouvrir1Click(Sender: TObject);
  var
    Dsn : TDessin;
begin
  if OpndlgDessin.Execute then
    begin
    NomFichier := OpndlgDessin.FileName;
    Dsn := TDessin.Create (NomFichier);
    Dessin.Free;
    Dessin := Dsn;
    invalidate;
    end;
end;
```

Remarque

Dans ce gestionnaire d'événement, nous sommes passés par une variable intermédiaire « `Dsn` » pour construire le nouveau dessin. Ceci nous permet de nous affranchir de toute erreur pouvant survenir dans la construction du nouveau dessin. Le code ci-dessus est dangereux si l'instanciation du dessin lève une exception :

```
procedure TForm1.Ouvrir1Click(Sender: TObject);
begin
  if OpndlgDessin.Execute then
    begin
    NomFichier := OpndlgDessin.FileName;
    Dessin.Free;
    Dessin := TDessin.Create (NomFichier);
    invalidate;
    end;
end;
```

En effet, en cas d'erreur, nous aurions détruit l'objet « `Dessin` » et celui-ci n'aurait pas été reconstruit. L'événement « `OnPaint` » partirait alors en erreur.

Pratique

Ajouter un champ « `NomFichier` » de type « `string` » à la fenêtre :

- Lors de la construction de la fiche « `NomFichier = '' ;` »
- Lors de la création d'un nouveau dessin remettez à 0 le nom de fichier.
- Lors de l'ouverture d'un fichier mémorisez le nom de fichier.
- Lors de l'enregistrement du dessin sous un nouveau nom, mémorisez le nom de fichier.
- Lors de l'enregistrement d'un fichier, si le nom de fichier existe, enregistrez, sinon appelez la méthode qui gère « `Enregistrer sous...` »

Pour gérer la création d'un nouveau dessin :

- Détruire le dessin actuel.
- Créer un nouveau dessin (créer une nouvelle instance).
- Demander un rafraîchissement de la fenêtre.

Pour gérer l'ouverture de fichier :

- Ouvrir la boîte de dialogue adéquate.
- Créer un nouveau dessin en le lisant depuis le flux.
- Détruire le dessin.
- Affecter le nouveau dessin au dessin.
- Demander un rafraîchissement de la fenêtre.

Pour gérer l'enregistrement du dessin :

- Enregistrer le dessin sous le nom « `NomFichier` ».

3.7.4 Valider la fermeture de la fiche

Dans cette section, essentiellement pratique, nous allons analyser l'état du dessin : « *A-t-il été modifié ?* » avant d'entreprendre un certain nombre d'actions, entre autres, avant d'autoriser la fermeture de la fiche.

Pratique

Ajoutez un champ « `Changement` » de type « `boolean` » dans la fiche pour vérifier si le dessin a changé avant d'accepter la destruction de l'ancien dessin ou avant de charger un nouveau dessin :

- Lors de la construction de la fiche « `Changement = false ;` ».
- A chaque fois que quelque chose est ajouté au dessin, « `Changement = true ;` ».
- Une fois qu'un nouveau dessin a été créé ou lu depuis un fichier, « `Changement = false ;` ».
- Avant de créer un nouveau dessin (commande « *Fichier / Nouveau* », testez l'état de la variable « `Changement` ». Si celle-ci est vraie, nous allons demander confirmation à l'utilisateur de confirmer l'opération avec un code tel que celui-ci :

```
procedure TForm1.Nouveau1Click(Sender: TObject);
begin
  if Changement then
    if MessageDlg (
      'Le dessin a changé, confirmez l''opération',
      mtConfirmation,
      [mbOk, mbCancel],
      0)
    = mrOk then
      begin
      Dessin.Clear;
      Changement := false;
      invalidate;
      end;
end;
```

La fonction « `MessageDlg` » permet d'afficher une boîte de dialogue avec un message et divers boutons — La valeur de retour de cette fonction indique le bouton qui a été utilisé pour fermer la boîte de dialogue. Cliquez sur la touche F1 alors que le curseur est sur le mot « `MessageDlg` » afin de faire apparaître l'aide sur cette fonction :

- Premier paramètre : le texte du message à afficher.
- Deuxième paramètre : un code indiquant la nature de la boîte de dialogue (confirmation, erreur…) — « mtConfirmation » indiquant qu'il s'agit d'un message de confirmation — Ce paramètre pilote l'icône affichée dans la boîte de dialogue.
- Troisième paramètre : un tableau contenant les codes des boutons à afficher (« `mbOK` » = le bouton « `OK` », « `mbCancel` » = le bouton « `Annuler` »).
- Quatrième paramètre : utilisé uniquement si un système d'aide est lié à l'application.
- La valeur de retour identifie le bouton utilisé : « `mrOk` » ou « `mrCancel` »

Pratique

Il existe en fait plusieurs endroit où on doit vérifier si le dessin a changé avant d'entreprendre une action :

- Avant d'ouvrir un dessin.
- Avant de créer un nouveau dessin.
- Avant de fermer la fenêtre.

Nous allons donc créer une nouvelle méthode privée : « `VerifierChangement` » qui pourra être utilisée à chacun de ces endroits :

```
type
  TForm1 = class(TForm)
    {...}
  private
    { Déclarations privées}
    {...}
    function VerifierChangement : boolean;
    {...}
  end;
{...}

function TForm1.VerifierChangement: boolean;
begin
  if Changement then
    if MessageDlg (
      'Le dessin a changé, confirmez l''opération',
      mtConfirmation,
      [mbOk, mbCancel],
      0)
    = mrOk then
      begin
      Changement := false;
      end;
  Result := not Changement;
end;
```

Le code de la création d'un nouveau dessin peut alors devenir :

```
procedure TForm1.Nouveau1Click(Sender: TObject);
begin
  if VerifierChangement then
    begin
    Dessin.Clear;
    invalidate;
    end;
end;
```

Nous allons voir ici comment autoriser la fermeture de la fiche. Il suffit d'intercepter l'événement « `OnCloseQuery`[24] » de la fiche :

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
  CanClose := VerifierChangement;
end;
```

Cet événement intègre un paramètre « `CanClose`[25] » qui valide ou non la fermeture de la fiche.

Pratique

Réalisez tous les tests de cette section.

[24] A traduire par « *A la demande de fermeture* »

[25] A traduire par « *Pouvoir fermer* »

4. LA GESTION DE FICHES DANS UN PROJET

Dans ce chapitre nous allons aborder diverses techniques permettant de gérer fenêtres et boîtes de dialogues.

4.1 LA GESTION DE FENETRES ET DE BOITES DE DIALOGUE

4.1.1 Certaines propriétés clefs des contrôles

Les fenêtres comme tous les contrôles possèdent un certain nombre de propriétés clefs à connaître :

- « `Name`[26] » : le nom que le programmeur affecte au composant. Ce nom se retrouve tel quel dans le source. Nous vous rappelons qu'une habitude des programmeurs *Delphi* consiste à préfixer le nom d'un composant par une abréviation rappelant la classe dont il est issu.
- « `Left`[27] » : la position gauche (en nombre de pixels) du composant par rapport au bord gauche du composant qui le contient. Dans le cadre d'une fenêtre c'est la position la fenêtre par rapport à l'écran.
- « `Top`[28] » : la position du haut (en nombre de pixels) du composant par rapport au bord supérieur de la partie utilisable du composant qui le contient. Dans le cadre d'une fenêtre c'est la position de la fenêtre par rapport à l'écran.
- « `Width`[29] » : la largeur du composant en nombre de pixels.
- « `Height`[30] » : la hauteur du composant en nombre de pixels.
- « `Caption` » : certains composants possèdent une propriété « *caption* » permettant à l'utilisateur d'identifier ce composant grâce à un libellé. Dans le cadre des fenêtres il s'agit de la barre de titre.

[26] A traduire par « *Nom* »
[27] A traduire par « *Gauche* »
[28] A traduire par « *Haut* »
[29] A traduire par « *Largeur* »
[30] A traduire par « *Hauteur* »

4.1.2 L'apparence d'une fenêtre

Diverses propriétés permettent de gérer l'apparence d'une boîte de dialogue. A cause de la nature de Windows, l'impact de certaines propriétés n'est pas évident lorsque l'on en regarde le nom. Voici un sommaire des propriétés clefs servant à gérer cette apparence :

- « `BorderStyle` » : Le nom de cette propriété signifie littéralement le « *style du bord* ». Cependant cette propriété influe non seulement sur le style du bord, mais aussi sur l'apparence et le comportement de la fenêtre. Voici la liste des valeurs possibles pour cette propriété :
  - « `bsSizable`[31] » : « *Style de bord dimensionnable* » : ceci convient pour les fenêtres normales de Windows qui peuvent être dimensionnées. Lorsque la souris passe sur un bord ou sur un coin elle change d'apparence et l'utilisateur peut changer la taille de la fenêtre.
  - « `bsDialog` » : « *Style de bord boîte de dialogue* » : ceci définit une boîte de dialogue : elle ne peut être redimensionnée, elle ne peut être transformée en icône ni agrandit en plein écran. Aucun menu principal ne peut être inséré dans une boîte de dialogue.
  - « `bsNone` » : « *Style de bord — Aucun* » : ceci définit une fenêtre sans bord (ni barre de titre) — Ceci convient particulièrement bien à un écran d'accueil qui est affiché au démarrage d'une application.
  - « `bsSingle`[32] » : « *Style de bord simple* » : ceci définit des fenêtres marquées par un bord simple. Ces fenêtres possèdent un titre, elles peuvent être mises en plein écran ou transformée en icône mais ne peuvent être redimensionnées.
  - « `bsToolWindow` » : « *Style de bord barre d'outils* » : le titre est affiché en plus petit, la fenêtre ne peut être dimensionnée, agrandie ou mise en icône.
  - « `SizeToolWin` » : « *Style de bord barre d'outils dimensionnable* » : le titre est affiché en plus petit, la fenêtre peut être dimensionnée, elle ne peut être agrandie ou mise en icône.

Remarque

Lorsque vous êtes en conception dans une fiche, lorsque vous saisissez du texte au clavier, celui-ci affecte directement la propriété active dans l'inspecteur d'objet.

Pour sélectionner une propriété dans une liste de constantes, vous pouvez commencer à introduire au clavier le début de la constante que vous désirez. *Delphi* recherchera la première valeur correspondante.

Pratique

Dans une nouvelle application testez successivement les diverses valeurs possibles pour la propriété « `BorderStyle` ».

---

[31] Sous les versions 3.x de *Windows*, ces fenêtres possédaient un encadrement marqué par une double ligne.

[32] Sous les versions 3.x de *Windows*, ces fenêtres possédaient un encadrement marqué par une ligne simple.

- « `BorderIcons` » : Cette propriété contient un ensemble de valeurs permettant de déterminer les icônes présentes dans la barre de titre de la fenêtre. Voici les codes des icônes qui peuvent être présentes dans la barre de titre d'une fenêtre :
  - « `biSystemMenu` » : Le menu système de la fenêtre sera présent, l'utilisateur pourra y accéder avec les touches Alt+<espace> ou avec la souris.
  - « `biMinimize` » : La fenêtre pourra être transformée en icône. Une icône est présente pour permettre de le faire dans la barre de titre. Si cette icône n'est pas présente dans la barre de titre, la fenêtre ne pourra pas être réduite.
  - « `biMaximize` » : La fenêtre pourra être agrandit pour prendre tout l'écran. Une icône est présente pour permettre de le faire dans la barre de titre. Si cette icône n'est pas présente dans la barre de titre, la fenêtre ne pourra pas être agrandie.
  - « `biHelp` » : Si un système d'aide contextuel a été écrit pour l'application, le bouton « Aide » permettra de sélectionner un composant dans la fenêtre et de charger la documentation le concernant.

    Remarque

    Lorsque dans l'inspecteur d'objet une propriété est préfixée d'un plus, « `+` », double cliquez sur ce signe afin de développer la propriété et pouvoir l'éditer. Lorsque la propriété est développée, un signe moins, « `-` » apparaît ; double cliquez sur celui-ci pour « *replier* » la propriété.

    Pratique

    Testez successivement les diverses valeurs possibles pour la propriété « `BorderIcons` ».
- « `WindowState`[33] » : Cette propriété traduit l'état normal (« `nsNormal` »), agrandi (« `wsMaximized` ») ou réduit en icône (« `wsMinimized` ») d'une fenêtre.

  Pratique

  Placez 3 boutons dans une fenêtre : « Normal », « Agrandie » et « Réduite ». Dans les gestionnaires d'événement « `OnClik` » de ces boutons pilotez l'état « `WindowState` » en conséquence.

[33] A traduire par « *Etat de la fenêtre* ».

- « `Position` » : Cette propriété codifie la position qu'occupera la fenêtre lors de sa première ouverture :
  - « `poDesigned` » : « *Position de conception* » : la fenêtre sera toujours créée avec les coordonnées « `Left` », « `Top` », « `Width` » et « `Height` » définie en mode conception.
  - « `poDefault` » : « *Position par défaut* » : la fenêtre sera ouverte en utilisant la taille et la position générées par défaut par *Windows*. D'une ouverture sur l'autre les fenêtres sont légèrement décalées vers le bas et vers la droite.
  - « `poDefaultPosOnly` » : « *Seulement la position par défaut* » : la fenêtre sera créée avec la taille « `Width` » et « `Height` » et avec la position calculée par *Windows*.
  - « `poDefaultSizeOnly` » : « *Seulement la taille par défaut* » : la fenêtre sera créée avec les coordonnées « `Left` », « `Top` » et la taille calculée par *Windows*.
  - « `poDesktopCenter` » : « *Au centre du bureau* »
  - « `poScreenCenter` » : « *Au centre de l'écran* » — Idéal pour une boîte de dialogue.

    Pratique

    Testez successivement les diverses valeurs possibles pour la propriété « `Position` » — Lancez plusieurs fois l'application avec la même valeur pour en voir l'influence.
- « `FormStyle` » : le « *style de fiche* » peut prendre les valeurs suivantes :
  - « `fsNormal` » : style normal.
  - « `fsStayOnTop` » : style « *Rester en avant plan* » : Lorsque l'application est active, la fiche reste en avant plan même si elle n'est pas active.
  - « `fsMDIForm` » et « `fsMDIChild` » : Ces propriétés seront examinées lors de l'étude de l'interface à documents multiples.

4.1.3 Les icônes

Lorsqu'un programme tourne, une icône et un titre d'application sont affichés dans la barre de tâche.

Pour paramétrer ceux-ci, utilisez la commande « *Projet / Options* » et allez dans l'onglet « *Application* » :

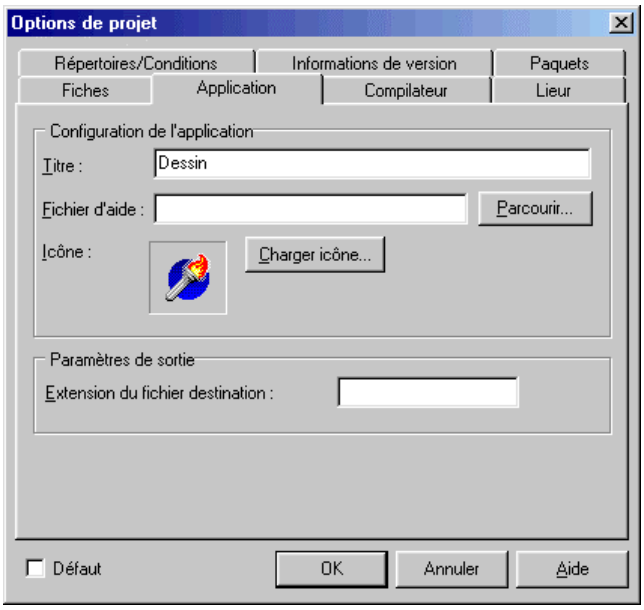

Figure 17. Le paramétrage de l'icône et du titre de l'application

Dans la zone « *Titre* », insérez le nom de l'application.

Vous pouvez cliquer sur le bouton « *Changer d'icône* » pour charger une nouvelle icône. Celle-ci sera mise dans la barre de tâches. Vous trouverez des icônes d'exemple dans le répertoire « *Program Files\Fichiers commun\Borland shared\Images\Icons* ».

Remarque

La commande « *Outils / Editeur d'images* » lance un outil capable entre autres de créer des icônes.

Notez qu'une icône peut exister en deux définitions :32x32 pixels ou 16x16 pixels (Si une de ces définitions n'existe pas dans l'icône, *Windows* la créera à partir de l'autre). Notez aussi que les icônes possèdent une couleur transparente.

L'icône chargée est mémorisée dans le fichier « `*.res` » de l'application.

Remarque

Examinez le source principal d'une application après avoir modifié les paramètres de cette boîte de dialogue. Vous y trouverez une ligne telle que :

```
Application.Title := 'Dessin';
```

Une fenêtre peut aussi avoir une icône personnelle. Pour ceci, éditez la propriété « `Icon` » de la fenêtre en cliquant sur les points de suspension.

Cliquez ensuite sur le bouton « `Charger…` » de la boîte de dialogue qui apparaît.

Pratique

Associez une icône et un titre à l'application de dessin que nous avons conçue au chapitre précédent. Associez aussi cette icône à la fenêtre de dessin (Il n'est normalement pas nécessaire d'associer des icônes aux boîtes de dialogue qui sont lancées depuis la fenêtre de dessin).

4.2 L'ACTIVATION D'UNE FENETRE

Trois méthodes clefs permettent de gérer l'activation d'une fenêtre :

- « `Show` » : Montre la fenêtre si elle était invisible (le même résultat peut être atteint en manipulant la propriété « `Visible` » et en lui affectant la valeur « `true` »).

  Lorsque plusieurs fenêtres ont été rendues visibles par cette technique, l'utilisateur peut circuler d'une fenêtre à l'autre.

  Lorsque la fenêtre est rendue visible, le programme se poursuit immédiatement en exécutant l'instruction qui suit cette instruction « `Show` » (sans attendre la fermeture de la fenêtre).
- « `Hide` » : Cache la fenêtre si elle était visible (le même résultat peut être atteint en manipulant la propriété « `Visible` » et en lui affectant la valeur « `false` »).
- « `ShowModal` » : Affiche la fenêtre et lui donne le contrôle total des événements d'entrée sortie jusqu'à ce que la fenêtre soit refermée. A la fermeture d'une fenêtre modale, la fonction « `ShowModal` » récupère un code permettant d'identifier la manière dont la fenêtre a été fermée, puis le programme se poursuit avec l'instruction qui suit cette instruction « `ShowModal` ».

Pratique

Vérifiez la différence entre « `Show` » et « `ShowModal` » : Créez une application qui contient 3 fenêtres (créez les deux fenêtres supplémentaires avec la commande « *Fichier / Nouvelle fiche* »).

Dans la fenêtre principale mettez deux boutons : « *Non modal* » et « *Modal* ».

Dans le gestionnaire de l'événement « `OnClick` » du bouton « *Non modal* » activez la deuxième fenêtre avec la méthode « `Show` » (« `Form2.Show ;` »).

Dans le gestionnaire de l'événement « `OnClick` » du bouton « *Modal* » activez la troisième fenêtre avec la méthode « `ShowModal` » (« `Form3.ShowModal ;` »).

Les fenêtres non modales

Les fenêtres non modales ne sont pas synchronisées les unes par rapport aux autres. L'utilisateur peut naviguer entre celles-ci.

Vous pouvez fermer une fenêtre non modale par le code en appelant sa méthode « `Close` ». Un processus de fermeture de fenêtre est déclenché dans le système et les événements « `OnCloseQuery` » et « `OnClose` » sont déclenchés.

Cependant, dans son comportement par défaut, le programme se contente de cacher la fenêtre. Ceci fait que l'objet qui gère la fenêtre n'est pas détruit. Vous pourrez ultérieurement réafficher cette fenêtre en lui appliquant la méthode « `Show` ».

Les fenêtres modales

Une boîte de dialogue est en général affichée en utilisant la méthode « `ShowModal` ».

Le programme qui affiche une boîte de dialogue ou une fenêtre modale est en général intéressé par l'analyse de la manière dont la fenêtre a été fermée (Par exemple, on n'entreprendra pas les mêmes actions si la fermeture s'est faite par le bouton « *OK* » ou le bouton « *Annuler* »).

Pour fermer une fenêtre modale par le code, vous modifiez le contenu de sa propriété « `ModalResult` » pour lui attribuer une valeur non nulle. Auquel cas le processus de fermeture de la fenêtre est déclenché et la valeur affectée à « `ModalResult` » est transmise à la fonction « `ShowModal` ».

Ce mécanisme peut être automatisé par les boutons. En effet, ceux-ci possèdent aussi une propriété « `ModalResult` » qui permet de piloter ce mécanisme.

Lorsque l'utilisateur clique sur un bouton d'une fenêtre modale, le bouton transmet automatiquement la valeur de sa propriété « `ModalResult` » dans le « `ModalResult` » de la fenêtre, et le mécanisme précédent est déclenché.

Remarque

Certaines constantes sont déjà déclarées dans la VCL pour identifier des valeurs standard de fermeture de fenêtre. Par exemple « `mrOK` », « `mrYes` », « `mrNo` », « `mrCancel` »…

Remarque

Comme pour les fenêtres non modales, la fermeture d'une fenêtre modale n'entraîne pas sa destruction. La fenêtre est juste cachée.

4.3 LA CREATION DYNAMIQUE DE BOITE DE DIALOGUE

4.3.1 Fiches auto créées versus création à la demande

Par défaut, *Delphi* instancie automatiquement toutes les fiches qui ont été attachées à un projet. Ceci convient parfaitement pour un projet de petite taille, mais n'est pas une bonne idée sur un projet de grande taille où il vaut mieux instancier fenêtres et boîtes de dialogue au moment où on en a besoin : 1/ on gagnera du temps de chargement au démarrage du programme 2/ on économisera les ressources systèmes.

L'onglet « *Fiches* » de la boîte de dialogue « *Projet / Options…* » permet de sélectionner quelle doivent être les fiches auto créées et quelle doivent être celles qui seront créées à la demande par le programmeur. Reprenez le projet précédent et appelez cette commande :

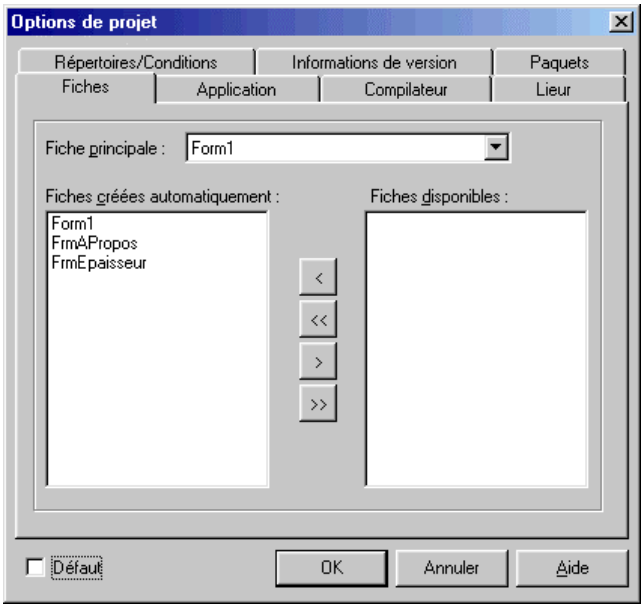

Figure 18. Le paramétrage des fiches auto créées

Les fiches dont le nom figure dans la colonne de gauche sont créées automatiquement. Les fiches dont le nom figure dans la colonne de droite devront être créées par le programmeur.

Dans notre exemple nous allons décider de créer manuellement les fiches « `FrmAPropos` » et « `FrmEpaisseur` ». Pour ceci.

- Sélectionnez la fiche « `FrmAPropos` ».
- Cliquez sur le bouton « `>` » pour la faire passer dans la colonne de droite.
- Sélectionnez la fiche « `FrmEpaisseur` ».
- Cliquez sur le bouton « `>` » pour la faire passer dans la colonne de droite.

Remarque

Si vous examinez le source du projet avant et après la modification de ce paramétrage, vous allez voir les différences suivantes :

Source avant :

```
program Texte;

uses
  Forms,
  fTexte in 'fTexte.pas' {Form1},
  Graphiques in 'Graphiques.pas',
  fAPropos in 'fAPropos.pas' {FrmAPropos},
  fEpaisseur in 'fEpaisseur.pas' {FrmEpaisseur};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TFrmAPropos, FrmAPropos);
  Application.CreateForm(TFrmEpaisseur, FrmEpaisseur);
  Application.Run;
end.
```

Source après :

```
program Texte;

uses
  Forms,
  fTexte in 'fTexte.pas' {Form1},
  Graphiques in 'Graphiques.pas',
  fAPropos in 'fAPropos.pas' {FrmAPropos},
  fEpaisseur in 'fEpaisseur.pas' {FrmEpaisseur};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Les instructions « `Application.CreateForm(...) ;` » ne sont présentes que pour les fiches auto créées.

4.3.2 L'instanciation d'une fiche

Tous les composants et, entre autres, les fiches possèdent un constructeur « `Create (AOwner : TComponent)` ».

Le paramètre de construction d'un composant, « `AOwner`[34] » fait référence au composant qui devient le propriétaire du composant ou de la fiche créée ; la fiche créée lui appartient.

Une des conséquences de la notion de propriétaire est que quand la propriétaire est détruit, les composants qu'il possède sont détruits automatiquement. Donc lorsqu'on affecte un composant à un propriétaire, on peut ne plus avoir à se soucier de sa destruction (si le cycle de vie du composant créé correspond au cycle de vie du composant propriétaire).

Une autre conséquence de la notion de propriétaire est que chaque composant tient à jour un tableau dynamique « `Components[...]` » dans lequel il référence les composants dont il est le propriétaire. Le fiche ou le composant créé apparaîtra alors dans le tableau « `Components[...]` » de son propriétaire. Une propriété « `ComponentCount` » permet de déterminer la taille de ce tableau (Les éléments sont stockés entre « `Components[0]` » et « `Components[ComponentCount-1]` »).

Les trois grandes manières de créer dynamiquement une fiche sont les suivantes :

- Affectation de la fiche créée à la fiche en cours :

```
procedure TForm1.Apropos1Click(Sender: TObject);
  var
    f : TForm2;
begin
  f := TForm2.Create (self);
  f.Show;
end;
```

- Affectation de la fiche créée à l'application :

```
procedure TForm1.Apropos1Click(Sender: TObject);
  var
    f : TForm2;
begin
  f := TForm2.Create (Application);
  f.Show;
end;
```

- Affectation de la fiche créée à aucun propriétaire :

```
procedure TForm1.Apropos1Click(Sender: TObject);
  var
    f : TForm2;
begin
  f := TForm2.Create (nil);
  f.ShowModal;
  f.Free;              // il est impératif de gérer la destruction
end;
```

[34] A Traduire par « *un propriétaire* »

4.3.3 La gestion de boîte de dialogue dynamique

Habituellement la gestion de boîte de dialogue dynamique obéit au scénario suivant :

- Création de la boîte de dialogue.
- Eventuellement faire les initialisations adéquates.
- Appeler « `ShowModal` » sur cette boîte de dialogue.
- Eventuellement faire les traitements adéquats en fonction de la réponse.
- Détruire la boîte de dialogue.

Voici un premier exemple pour la gestion de la boîte de dialogue « *A Propos* » :

```
procedure TForm1.Apropos1Click(Sender: TObject);
  var
    f : TFrmAPropos;
begin
  f := TFrmAPropos.Create (nil);
  f.ShowModal;
  f.Free;              // il est impératif de gérer la destruction
end;
```

Cependant cette technique doit être améliorée pour nous assurer que la boîte de dialogue soit détruite même si une exception surgit entre l'instanciation et la destruction. Protégeons le code :

```
procedure TForm1.Apropos1Click(Sender: TObject);
  var
    f : TFrmAPropos;
begin
  f := TFrmAPropos.Create (nil);
  try
    // éventuelles initialisations
    f.ShowModal;
    // éventuels traitements
  finally
    f.Free;
  end;
end;
```

Ou encore sans passer par la déclaration d'une variable de travail

```
procedure TForm1.Apropos1Click(Sender: TObject);
begin
  with TFrmAPropos.Create (nil) do
    try
      // éventuelles initialisations
      ShowModal;
      // éventuels traitements
    finally
      Free;
    end;
end;
```

Pratique

Intégrez cette gestion de la boîte de dialogue.

Réalisez une gestion semblable pour la boîte de dialogue de gestion de l'épaisseur du trait.

4.4 LA CREATION DYNAMIQUE DE FENETRE

Habituellement la création de fenêtre dynamique obéit aux deux scénarios suivants :

- La fiche n'a pas été créée, mais dès qu'une commande la crée, on la conserve dans l'environnement. Si elle est fermée, elle n'est pas détruite et on peut la réactiver.
- Lorsqu'on ferme la fiche, on veut aussi procéder à sa destruction.

4.4.1 Conserver les fiches créées à la demande

Dans le cadre du premier scénario, le problème se traite de la manière suivante dans le code qui crée la fiche :

```
procedure TForm1.Apropos1Click(Sender: TObject);
begin
  if not Assigned (Form2) then
    begin
    Form2 := TForm2.Create (Application);
    Form2.Show;
    end
  else
    begin
    // traitement adéquat
    end
end;
```

Dans ce scénario nous manipulerons en général la variable globale qui sert à identifier la fiche par défaut de cette classe.

La fonction « `Assigned` » nous permet de détecter si la fiche a été crée ou non. Si elle n'a pas été créée nous la créons et l'affectons à l'application pour qu'elle soit détruite automatiquement lorsque l'application se terminera, puis nous la rendons visible.

Par contre si la fiche existe, c'est le contexte du programme qui nous permettra de décider ce qu'il faut faire :

- Si la fiche a été cachée, nous pouvons l'activer avec « `Form2.Show ;` »
- Si la fiche est toujours visible mais qu'elle est en arrière plan, nous pouvons la faire passer en avant plan avec « `Form2.BringToFront ;` »

Pratique

Créez une nouvelle application avec une fiche principale, « `FrmMenu` » que nous appellerons un menu qui contient trois boutons « *Client* », « *Fournisseur* » et « *Facture* ».

Dans cette application nous aurons besoin de trois autres fiches : « `FrmClient` », « `FrmFournisseur` » et « `FrmFacture` ». Mettez un libellé ou des contrôles sur ces trois fiches pour illustrer les différences entre celles-ci.

Retirez ces trois fiches de la liste des fiches auto créées.

Dans le gestionnaire d'événement « `OnClick` » de chacun des boutons de la fiche principale, créez dynamiquement la fiche correspondante avec la technique de cette section.

4.4.2 Détruire les fiches créées à la demande

Dans ce contexte, l'utilisateur pourra ouvrir autant de fiche d'un même type qu'il le désire.

Nous ne pouvons pas écrire un code tel que :

```
procedure TForm1.Apropos1Click(Sender: TObject);
  var
    f : TForm2;
begin
  f := TForm2.Create (Application);
  f.Show;
  f.Free;          // code impossible
end;
```

En effet, la méthode « Show » n'attends pas que le fiche se ferme. Elle rend la fiche visible et le code continue immédiatement. La fiche est alors détruite avant que l'utilisateur ait une chance de la voir.

Le code doit donc absolument être :

```
procedure TForm1.Apropos1Click(Sender: TObject);
  var
    f : TForm2;
begin
  f := TForm2.Create (Application);
  f.Show;
end;
```

Il nous faut alors trouver une autre manière de détruire la fiche. Notez que la variable « `F` » est locale à la méthode et que nous perdons la trace de la fiche qui a été créée.

La réponse est dans l'événement « `OnClose` » défini dans la classe « `TForm2` » :

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin

end;
```

Cet événement récupère un paramètre « `Action` » du type « `TCloseAction` » qui détermine ce que doit devenir la fiche lors de sa fermeture :

- « `caHide` » : la fiche reste instancié mais est cachée (appel de la méthode « `Hide` ») — C'est la valeur par défaut du paramètre.
- « `caFree` » : la fiche est détruite.
- « `caMinmize` » : la fiche est transformée en icône.
- « `caNone` » : aucune action n'est entreprise. La fiche ne se ferme pas.

Le gestionnaire d'événement devient donc :

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Pratique

Revoyez l'exemple précédent avec cette technique. Remarquez que l'utilisateur peut utiliser autant de fiche client, fournisseur ou facture qu'il le désire.

4.5 L'INTERFACE A DOCUMENTS MULTIPLES

4.5.1 Le modèle d'application *MDI*

Les applications de type *MDI* (Multiple Document Interface) sous *Windows* sont constituées d'une fenêtre principale qui peut contenir plusieurs fenêtres.

Delphi peut générer un squelette d'application *MDI*, qu'il est alors facile de modifier. Pour ceci créez un nouveau projet de la manière suivante :

- Dans le menu choisissez « *Fichier / Nouveau …* ».
- Choisissez l'onglet « *Projets* ».
- Choisissez « *Application MDI* »
- Sélectionnez le répertoire dans lequel *Delphi* doit construire l'application.

Mise en pratique

Réalisez ce nouveau projet et exécutez-le pour vous rendre compte de ce que comporte déjà une application *MDI* générée par *Delphi*.

4.5.2 Intégration de la fenêtre de dessin

Nous allons ici intégrer et adapter la fenêtre de dessin au fonctionnement *MDI*.

- Avec l'option « *Fichier / Ajouter fichier* », intégrez le fichier source de la fiche (fichier *.pas).
- Avec l'option « *Fichier / Enregistrer sous* », sauvegardez cette fenêtre sous un nouveau nom afin d'en préserver l'original.
- Avec l'option « *Option / Projet / Fiches* », assurez-vous que cette fenêtre ne soit pas créée automatiquement au démarrage du programme.
- Visualisez la fiche de dessin dans l'espace de travail et grâce à l'*inspecteur d'objets*, Changez la valeur de la propriété `FormStyle` de `fsNormal` à `fsMDIChild`. Une fenêtre de style `fsMDIChild` peut être gérée dans un environnement *MDI*, pas une fenêtre `fsNormal` et vice versa.

La matière première est maintenant présente dans le projet, et nous allons pouvoir la travailler. Certaines de nos commandes doivent être déplacées (ou plutôt dupliquées) du menu de la fenêtre de dessin vers le menu de la fenêtre principale.

Mise en pratique

Créer une *méthode* **CreerMDIDessin** sur le modèle de **CreateMDIChild**, et appeler cette méthode depuis l'*événement* « *Fichier / Nouveau* ».

Remarquez, à l'exécution, que le *menu* de la fenêtre de dessin vient remplacer le menu originel dès qu'une fenêtre de dessin est ouverte — A terme toutes les commandes seront gérées depuis le menu de la fenêtre principal.

En attendant que le programme soit complètement terminé nous allons fusionner les deux *menus* (les mettre côte à côte). Modifiez les propriétés suivantes dans la fenêtre de dessin, avec l'*inspecteur d'objet* :

- **MainMenu.AutoMerge = False**; — Cette *propriété*, **AutoMerge**, autorise la fusion des menus avec le menu principal pour les applications qui ne sont pas MDI.
- **Fichier1.GroupIndex = 1 ;** — Cette *propriété*, **GroupIndex**, d'un item de menu détermine où cet item de menu de la fenêtre *MDI* doit être inséré dans le menu de la fenêtre principale. — Les items de menus de la fenêtre principale possédant le même **GroupIndex** que ceux de la fenêtre *MDI* sont remplacés par ceux de la fenêtre *MDI*.
  - Double cliquez sur l'icône du menu dans la fenêtre de dessin pour faire surgir le *concepteur de menu*.
  - Lors de la mise à 1 de la propriété **GroupIndex** du premier item de menu, cette même propriété passe à 1 pour tous les items qui suivent, pour respecter un ordre croissant de valeur.
- Et de la même manière, dans la fenêtre principale **Window1.GroupIndex = 2 ;** et **Help1.GroupIndex = 2 ;** — Ceci assurera que les options « Fenêtre » et « Aide » du programme principal resteront à droite, puis :
  - Déplacez alors le code du chargement de la fenêtre « *A propos* » dans la fenêtre principale (de manière à faire apparaître la fenêtre « A propos » même si aucune fenêtre de dessin n'est présente.
  - Supprimer l'item « *A propos* » de la fenêtre de dessin.

Notez aussi que lorsque vous fermez une fenêtre de dessin celle-ci reste sous forme d'icône en bas de la fenêtre principal, chose que ne font pas les fenêtres du châssis originel :

- Ouvrez la fenêtre **MDIChild** dans l'environnement de développement de *Delphi* et observez comment est géré l'événement **OnClose** :

```
procedure  TMDIChild.FormClose
           (Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

  - Le paramètre **Action** de cet événement détermine comment doit se comporter la fenêtre lors de la demande de fermeture — Les valeurs possibles sont : **caNone**, **caHide**, **caFree** et **caMinimize** (Pas d'action, cacher la fenêtre, la détruire ou la minimiser — ce dernier étant le comportement par défaut).
  - Gérez de la même manière **OnClose** pour la fenêtre de dessin.

Mise en pratique suite

Pour terminer l'application, il faut déplacer les autres éléments du menu « *Fichier* » de la fenêtre de dessin vers le menu « *Fichier* » de la fenêtre principale, puis détruire le menu fichier de la fenêtre de dessin :

- « *Fichier / Ouvrir* » :
  - Créer un deuxième *constructeur* pour la fenêtre de dessin ; constructeur auquel on passe un paramètre supplémentaire : le nom de fichier — Ouvrez le fichier dans le constructeur — Modifier le titre de la fenêtre dans le constructeur pour que le nom de fichier soit affiché. Le constructeur aura la forme

```
constructor TForm1.CreerDessin
     (AOwner : TComponent; Fichier : TFileName);
```

  - Modifier l'*événement* « *Fichier / Ouvrir* » en conséquence — Modifiez les *propriétés* de la *boîte de dialogue* **OpenDialog**, avec l'*inspecteur d'objets* afin que les bons filtres de fichiers et la bonne extension (*PTS*) soit présente.
  - Dans la fenêtre de dessin supprimer les items « *Fichier / Nouveau* », « *Fichier / Ouvrir* », les *événements* et les boutons correspondants.
- « *Fichier / Fermer* » : rien à faire

4.5.3 A quelle *classe* appartient un objet ?

Pour gérer l'option « *Fichier / Enregistrer* », ainsi que d'autres options (comme imprimer) de la fenêtre principale, nous rencontrons un problème :

Depuis la fenêtre principale, on peut retrouver la fenêtre fille active grâce à la *propriété* **ActiveMDIChild**. Cependant, dans un environnement *MDI*, rien ne garantit que cette fenêtre soit de la classe **TForm1**, en effet plusieurs fenêtres de différentes *classes* peuvent être présentes. Si la fenêtre est de type **TForm1**, nous pourrons lui appliquer les *méthodes* adéquates pour gérer l'enregistrement, sinon, il faudra utiliser d'autres *méthodes*.

L'opérateur `IS`

L'opérateur **IS** permet de déterminer si un objet appartient à une classe donnée :

```
Objet IS Classe
```

renvoie la valeur **TRUE** si l'objet appartient à la classe concernée, **FALSE** dans le cas contraire.

Pour tester si **ActiveMDIChild** fait partie de la *classe* **TForm1** nous pourrons alors utiliser :

```
IF ActiveMDIChild IS TForm1 THEN ...
```

L'opérateur AS

Lorsque nous avons déterminé que l'objet est de la *classe* adéquate, nous savons que nous pouvons lui appliquer les *méthodes* de cette *classe* ; cependant d'un point de vue syntaxique, le *Pascal Objet* ne nous laissera pas faire. Il faut procéder à un changement de type (*type casting*).

Avec les premières extensions du Pascal Objet de Borland, nous aurions pu écrire :

```
TForm1 (ActiveMDIChild).Methode
```

Le nom d'une *classe* suivi du nom de l'*instance* mis entre parenthèses demande à ce que cette *instance* soit temporairement considérée comme faisant partie de la *classe* concernée.

Une autre méthode consiste à utiliser l'opérateur **as** avec une forme :

```
Objet as Classe
```

Par exemple

```
with ActiveMDIChild as TForm1 do ...
```

Mise en pratique

Pour gérer

- « *Fichier / Enregistrer* » :
  - Créer une *méthode*, `function QuelFichier : TFileName ;`, qui demande le nom de fichier à la fiche de dessin (en toute logique le nom de fichier de la fiche de dessin devrait être un *champ privé* pour qu'on ne soit pas tenté de le modifier « à la sauvage » de l'extérieur — respect de l'*encapsulation*).
  - Créer une *méthode*, `TForm1.Enregistrer`, dans la fenêtre de dessin pour enregistrer le dessin.
  - Gérer l'événement enregistrer avec un code tel que celui-ci :

```
procedure TMainForm.FileSaveItemClick(Sender: TObject);
begin
if ActiveMDIChild = nil then exit;

if ActiveMDIChild is TForm1 then
  with ActiveMDIChild as TForm1 do
    begin
    if QuelFichier <> '' then
      Enregistrer
    else
      FileSaveAsItemClick(Sender);
    end;
end;
```

  - Supprimer dans la fiche de dessin le code lié à la gestion de la commande « *Fichier / Enregistrer* », l'option de menu et le bouton.

- « *Fichier / Enregistrer sous* » :
  - Créer une *méthode*, `TForm1.EnregistrerSous (Fichier) ;`, dans la fenêtre de dessin, pour enregistrer le dessin, et modifier le nom de fichier et le titre de la fenêtre.
  - Dans la fenêtre principale, placer un objet `SaveDialog1` de la classe `TSaveDialog` et le paramétrer en conséquence.
  - Intercepter l'événement « *Fichier enregistrer* » ; tester si la fenêtre est du type correct, charger le bon nom de fichier dans la boîte de dialogue, ouvrir la boîte de dialogue, si nécessaire appeler la *méthode* précédente d'enregistrement de la fiche fille.
  - Supprimer dans la fiche de dessin le code lié à la gestion de la commande « *Fichier / Enregistrer sous* » et l'option de menu.
- « *Fichier / Imprimer* » :
  - Procédez comme précédemment pour créer une *méthode* `Imprimer` dans la fenêtre de dessin.
  - Créer dans la fenêtre principale les options de menu « *Fichier / Imprimer* » et « *Fichier / Configuration de l'imprimante* », ainsi qu'un bouton Imprimer.
  - Gérer l'*événement* et le bouton imprimer comme précédemment.
  - Déplacer le code de la fenêtre de dessin vers la fenêtre principale pour gérer l'option « *Fichier / Configuration de l'imprimante* ».
  - Créer éventuellement un bouton pour gérer l'impression.
  - Supprimer code, option de menu et bouton de la fenêtre de dessin — Supprimer aussi le code et l'option liés à « *Fichier / Quitter* », puis supprimer le menu fichier. Il ne doit rester que le menu « Option » dans la fenêtre de dessin.

L'application de dessin *MDI* est maintenant terminée.

5. LES COMPOSANTS ET LES CONTROLES

5.1 LES CONTENEURS

Les *conteneurs* sont des contrôles (ou des objets) qui peuvent contenir d'autres contrôles (ou d'autres objets).

La fenêtre, `TForm`, est un conteneur, en effet elle contient des boutons, des zones de saisie… et toute sorte de contrôles.

Le panneau, `TPanel`, est aussi un conteneur.

Un menu, `TMenu`, est un conteneur d'un autre type, il contient des items de menu.

Comme vous le verrez dans la description des composants, d'autres contrôles sont des conteneurs.

5.2 L'INSTANCIATION DYNAMIQUE DES CONTROLES

5.2.1 La propriété « Parent »

Chaque composant possède une propriété « `Parent` » (à ne pas confondre avec la propriété « `Owner` ») qui détermine le contrôle dans lequel ce contrôle est affiché. La propriété « `Owner` » est en lecture seule, la propriété « `Parent` » est en lecture et en écriture. Ceci signifie que l'on peut déplacer un composant d'un conteneur vers un autre à l'exécution.

Par exemple :

- Créez une fiche qui contienne deux « `TPanel` ».
- Dans le premier « `TPanel` », déposez un « `TEdit` ».
- Dans la fiche placez deux boutons et écrivez les gestionnaires d'événement suivants :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Parent := Button1;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit1.Parent := Button2;
end;
```

A l'exécution en fonction du bouton sur lequel vous cliquez, vous voyez la zone d'édition se déplacer d'un panneau vers l'autre.

Remarque

Chaque contrôle tient à jour un tableau dynamique « `Controls[...]` » et une propriété « `ControlCount` » permettant de retrouver tous les composants dont un contrôle est le parent.

De la même manière, chaque composant tient à jour un tableau dynamique « `Components[…]` » et une propriété « `ComponentCount` » permettant de retrouver tous les composants dont ils sont les propriétaires. (Notez que comme un contrôle est aussi un composant, il possède aussi ces propriétés).

5.2.2 L'instanciation des contrôles

Lorsque vous instanciez dynamiquement un contrôle, non seulement il faut préciser le propriétaire dans le constructeur, mais en plus il faut affecter quelque chose à sa propriété parent pour qu'il devienne visible.

Pratique

Créez dynamiquement un bouton, et affichez-le dans la fenêtre où vous l'avez créé.

5.3 LES *COLLECTIONS*

Une *collection* est un objet qui permet de stocker une série d'objets. Nous avons déjà vu la *classe* `TList` qui est une sorte de *collection*. Un objet `TList` se comporte comme un tableau dynamique de pointeur, et peut donc servir à garder la trace d'une liste d'objet.

Avec *Delphi 5*, nous avons une autre classe, `TObjectList` beaucoup plus pratique pour maintenir une collection d'objets. En effet un objet `TObjectList` se comporte comme un tableau dynamique de `TObject`. Il possède un état et des comportements semblables à un `Tlist`. Un `TObjectList` possède une caractéristique supplémentaire. Il peut être déclaré comme étant le propriétaire des objets qu'il contient. Auquel cas lorsqu'un objet et retiré de la liste ou lorsque la liste est détruite, l'objet ou les objets contenus sont aussi détruits.

De la même manière nous trouvons la *classe* `TStrings` et sa sous-classe `TStringList` qui permettent de mémoriser une liste de chaînes de caractères (et qui permet aussi, accessoirement d'associer un objet à ces chaînes). Le stockage de chaînes dans une `TStrings` ressemble à ce que nous avons déjà vu pour une `TList`.

De nombreux *composants* manipulent des tableaux de chaînes de caractères grâce à des `TStrings`.

Lorsque vous devrez manipuler par le code des objets compatibles avec des `TStrings`, vous utiliserez la sous-classe `TStringList`.

5.4 LES *COMPOSANTS* ET LES *CONTROLES*

Voici une liste récapitulative des principaux *composants* et *contrôles* de *Delphi*. La section suivante de ce chapitre abordera des exemples et des exercices concernant ces composants et contrôles :

5.4.1 Les *composants* standards

Figure 19. Les composants standard de Delphi 5

- `TFrame` : *Cadre* — Un « *frame* » est une sorte de méta-objet (objet géant) que vous avez déjà fabriqué afin de le réutiliser à divers endroits de votre ou de vos applications. Il est constitué d'un ensemble de composants, de contrôles et de lignes de code destinés à être *réutilisés*. L'utilisation des « frames » mérite une section à part à la quelle nous vous renvoyons.
- `TMenu` : *Menu* — principalement attaché en haut d'une fenêtre.
- `TPopupMenu` : Menu surgissant pouvant être associé au clic droit sur un composant.
- `TLabel` : zone dans laquelle afficher du texte.
- `TEdit` : Editeur, zone dans laquelle saisir et corriger du texte.
- `TMemo` : Mémo, zone de manipulation de texte semblable à un `TEdit`, mais pouvant travailler sur plusieurs lignes (différence de gestion de la touche ↵).
- `TButton` : Bouton (Voir aussi les `TBitBtn` des composants supplémentaires).
- `TCheckBox` : Case à cocher.
- `TRadioButton` : Bouton radio — dans un groupe de boutons radio, seul l'un d'entre eux peut être coché — Plutôt que d'utiliser des `TRadioButton`, vous devriez chercher à utiliser des `TRadioGroup` beaucoup plus faciles à mettre en œuvre.
- `TListBox` : Liste de chaînes de caractères affichées à l'écran pour permettre le choix d'une ou plusieurs chaînes. Eventuellement un ascenseur peut apparaître sur la droite. La liste peut ou non être triée.
- `TComboBox` : Combinaison d'un `TEdit` et d'une `TListBox` — A Priori la liste est cachée et n'apparaît que lorsque c'est demandé par l'utilisateur. Plusieurs modes de fonctionnement sont disponibles en fonction de la propriété `Style`.
- `TScrollBar` : Barre de défilement (Ascenseur ou translateur).
- `TGroupBox` : Cadre permettant de regrouper plusieurs *composants* et de les considérer comme un macro *composant*.

- `TRadioGroup` : Sorte de `TGroupBox` pour les `TRadioButton`.
- `TPanel` : Panneau (cadre et fond). Souvent utilisé pour créer des *barres d'outils*.
- `TActionList` : Liste d'actions. Une action permet de regrouper en un seul point un élément d'interface utilisateur (description d'un item de menu, d'une icône, d'un bouton). Cet élément d'interface pourra alors être dispatché en divers endroit dans le programme. Nous vous conseillons d'utiliser ces `Actions`, plutôt que de programmer directement dans le menu, l'icône le bouton. En effet, en programmation, il arrive fréquemment les choses suivantes : d'une part, plusieurs contrôles permettent d'arriver à la même commande, d'autre part, il arrive dans le processus de développement que l'on modifie son interface (qu'une action à l'origine était déclenchée par un menu, et maintenant par une icône). Les `Actions` résolvent pratiquement ces phénomènes.

5.4.2 Les *composants* supplémentaires

Figure 20. Les composants supplémentaires

- `TBitBtn` : Sorte de `TButton` (Bouton), dans lequel on peut charger une image graphique à côté du texte du bouton (Bouton graphique).
- `TSpeedButton` : Sorte de `TButton` (Bouton) dans lequel on peut charger une image graphique ou une icône à la place du texte — Ce type de bouton est souvent utilisé pour faire des *barres d'outils*.
- `TMaskEdit` : Sorte de TEdit auquel est associé un masque de saisie pour contrôler la validité du texte saisi.
- `TStringGrid` : Grille (tableau) de chaînes de caractères.
- `TDrawGrid` : Grille (tableau) de dessins.
- `TImage` : Image (Bitmap ou autre).
- `TShape` : formes graphiques prédéfinies (Cercle, rectangle, etc...).
- `TBevel` : Sorte de `TPanel` (panneau) au bord en relief.
- `TScrollBox` : Boîte déroulante — Petite fenêtre avec un ascenseur et un translateur.
- `TCheckedListBox` : Sorte de `TListBox` où chaque item peut être coché.
- `TSplitter` : Composant permettant de définir une limite mobile entre deux autres composants (Par exemple entre un tableau et une zone de texte — l'utilisateur pourra alors déplacer cette limite avec la souris).
- `TStaticText` : Zone dans laquelle afficher du texte, semblable à un `TLabel`. A moins que vous n'en ayez expressément besoin, utilisez plutôt un `TLabel`, le `TStaticText` monopolisant plus de ressources dans *Windows*.
- `TControlBar` Barre de contrôle: Espace pouvant accueillir des barres d'outils. Ces barres d'outils (voir `TToolBar`) peuvent alors être détachées ou rattachées de la barre de contrôle.

- `TApplicationEvents` : Une application est pilotée par un objet principal, `Application` de la classe `TApplication`. L'objet `Application` n'est pas présent dans l'environnement graphique de *Delphi*, mais n'est accessible que par le code. Un composant `TapplicationEvents` permet aussi d'accéder aux événements de l'application depuis l'interface graphique.
- `TChart` : Principal élément constitutif des graphes qui ne sont pas issues des données d'une base de données.

5.4.3 Les *boîtes de dialogue*

Figure 21. Les boîtes de dialogue

- `TOpenDialog` : *Boîte de dialogue* d'ouverture de fichier.
- `TSaveDialog` : *Boîte de dialogue* d'enregistrement de fichier.
- `TOpenPictureDialog` : *Boîte de dialogue* d'ouverture de fichier graphique (pour ouvrir les fichiers graphiques standard — Fonctionne comme une `TOpenDialog` et possède en plus une zone de prévisualisation).
- `TSavePictureDialog` : *Boîte de dialogue* d'enregistrement de fichier graphique. (Fonctionne comme une `TSaveDialog` et possède en plus une zone de prévisualisation).
- `TFontDialog` : *Boîte de dialogue* de choix de caractère (police, taille, ...).
- `TColorDialog` : *Boîte de dialogue* de choix de couleur.
- `TPrintDialog` : *Boîte de dialogue* d'impression (nombre de pages,...).
- `TPrinterSetupDialog` : *Boîte de dialogue* de configuration de l'imprimante.
- `TFindDialog` : *Boîte de dialogue* de paramétrage de recherche de texte.
- `TReplaceDialog` : *Boîte de dialogue* de paramétrage de « rechercher et remplace ».

5.4.4 Les *composants* Win32

Win32 désigne un ensemble de fonctions permettant à une application de dialoguer avec *Windows*. Ces fonctions ont été ajoutées à Windows lors du passage au mode 32 bits de *Windows*.

Figure 22. Les composants Win32

- `TTabControl` : Contrôle à onglets destiné à gérer une série d'onglet pouvant se trouver sur un des côtés d'une fiche ou d'un composant. Pour ce composant, considérez le `TTabControl` comme une liste de boutons adjacent à une zone dans laquelle vous pouvez placer divers composants (zone dont le comportement ressemble à celui d'un `TPanel`).
- `TPageControl` : Contrôle de pages : Contrôle à onglets multipage (à chaque fois que vous cliquez sur un onglet, vous faite apparaître une page différente contenant les composants associés à l'onglet sur lequel vous avez cliqué).

- `TImageList` : Liste d'images : Composant contenant une liste d'icônes (toutes à priori de la même taille) ; cette liste d'icônes peut alors être associée à un menu, une barre d'outils… afin d'attribuer une icône à chaque item de menu ou de la barre d'outils.
- `TRichEdit` : Editeur de texte enrichi (*RTF* ou « Rich Text Format »). Zone de manipulation de texte semblable à un `TMemo`, permettant aussi de mettre en forme ce texte comme dans un éditeur de texte (Changement de police, différents styles de paragraphes…).
- `TTrackBar` : Glissière. Contrôles ressemblant à un réglage à glissière d'une table de mixage. L'utilisateur peut régler la position d'un curseur en face d'une échelle graduée.
- `TProgressBar` : Barre de progression. Voir aussi le composant `TGauge`.
- `TUpDown` : Flèches haut/bas. Voir aussi `TspinButton`.
- `THotKey` : Editeur de raccourci de clavier. Sorte de `TEdit` permettant de saisir des raccourcis de clavier.
- `TAnimate` : Dessin animé.
- `TDateTimePicker` : Editeur de date et/ou d'heures. Une liste déroulante permet de choisir la date ou l'heure dans un calendrier.
- `TMonthCalendar` : Calendrier mensuel.
- `TTreeView` : Vue arborescente, semblable à celle du panneau de gauche de l'explorateur de fichiers de *Windows*.
- `TListView` : Vue de type liste, semblable à ce celle du panneau de droite de l'explorateur de Windows.
- `THeaderControl` : Entête. Ce contrôle gère un ensemble d'en-têtes de colonne redimensionnable.
- `TStatusBar` : Barre d'état (en général en bas de la fenêtre principale d'une application).
- `TToolBar` : Barre d'outils.
- `TCoolBar` : Barre de contrôle « fantaisie ». Barre d'outils comme celle qu'on a l'habitude de voir dans les outils bureautique.
- `TPageScroller` : Zone défilement horizontal essentiellement destinée à contenir une barre d'outils afin de pouvoir la faire défiler à droite ou à gauche si on a plus d'outils que de place disponible.

5.4.5 Les *composants* systèmes

Figure 23. Les composants système

- `TTimer` : Gestion du temps, de délais et/ou d'alarme.
- `TPaintBox` : Fenêtre de dessin.
- `TMediaPlayer` : Fenêtre d'exécution multimédia.
- `TOLEContainer` : Conteneur *OLE*.
- `TDDEClientConv`, `TDDEClientItem`, `TDDEServerConv` et `TDDEServerItem` : Diverses classes pour gérer le protocole *DDE*.

5.4.6 Les exemples de *composants*

Ces exemples de *composants* ont été joints à *Delphi* pour illustrer comment un programmeur peut créer de nouveaux *composants* (stage de niveau 3). Plusieurs de ces *composants* peuvent avoir une utilité immédiate :

Figure 24. Les exemples de composant

- `TGauge` : Jauge (indicateur de progression avec un pourcentage). Voir aussi le composant `TProgressBar`.
- `TColorGrid` : Grille de sélection de couleur.
- `TSpinButton`. Voir aussi `TUpDown`.
- `TSpinEdit` : Editeur (`TEdit`) pour la saisie d'une valeur numérique avec un `SpinButton` pour augmenter/diminuer la valeur.
- `TDirectoryOutline` : Liste hiérarchisée (`TOutline`) des répertoires.
- `TCalendar` : Calendrier.
- `TIBEventAlrter` : Composant lié à l'utilisation de la base de données *Interbase* (non étudié dans ce manuel).

6. LA MISE EN ŒUVRE DES COMPOSANTS

6.1 QUELQUES PROPRIETES CLEFS COMMUNES A TOUS LES CONTROLES

La plupart des contrôles ont en commun un certain nombre de propriété qui pourront soit être manipulées depuis l'inspecteur d'objet, soit depuis le code. Voici un résumé des principales propriétés :

- `Name` : « Nom ».
- `Top`, « Haut » et `Left`, « Gauche ». Position du contrôle.
- `Height`, « Hauteur » et `Width`, « Largeur ». Taille du contrôle.
- `Color` : « Couleur ». La couleur de fond du contrôle.
- `Enabled` : « Activé ». Certains contrôles peuvent être manipulés par l'utilisateur (par exemple une zone de saisie). Lorsque ces contrôles sont inactif, l'utilisateur ne peut les manipuler (ils sont alors généralement présenté dans un état grisé).
- `Font` : « Police de caractères ».
- `Hint`, « Conseil » et `ShowHint`, « Voir conseil ». Lorsque `ShowHint` vaut `true`, le texte indiqué dans `Hint` est affiché dans une « bulle » jaune lorsque la souris passe sur le contrôle.
- `ReadOnly` : « Seulement en lecture ».
- `TabStop`, « Arrêt de tabulation » et `TabOrder`, « Ordre de tabulation ». L'utilisateur peut passer d'un contrôle à l'autre grâce à la touche ⏎ pour tous les contrôles dont `TabStop` est à `true`. `TabOrder` indique dans quel ordre.
- `Visible` : « Visible ».

6.2 LES ALIGNEMENTS ET LA PRESENTATION

6.2.1 La propriété Align

La plupart des contrôles possèdent une propriété `Align` qui détermine comment ce contrôles doit être présenté par rapport à son *parent* (contrôle dans lequel il est affiché). Les valeurs possibles sont les suivantes :

- `alNone` : Pas de contrainte d'alignement.
- `alTop` : Le contrôle doit rester en haut de son parent.
- `alBottom` : Le contrôle doit rester en base de son parent.
- `alLeft` : Le contrôle doit rester à gauche de son parent.
- `alRight` : Le contrôle doit rester à droite de son parent.
- `alCenter` : Le contrôle doit rester dans la partie centrale de son parent.

Voici un exemple où plusieurs panneaux (`TPanel`) sont disposé à l'intérieur d'une fenêtre ; la position de ces panneaux est contrôlée par la propriété `Align` ; ainsi lorsque la fenêtre change de taille, la position des panneaux reste cohérente :

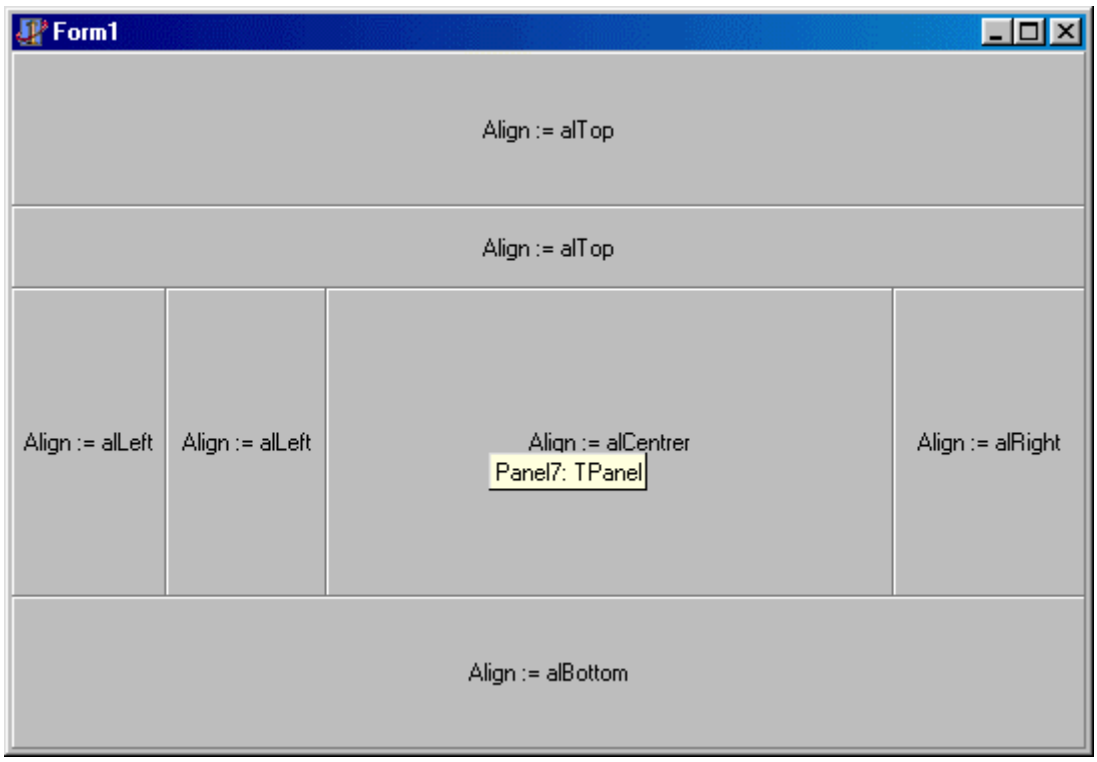

Figure 25. La propriété Align

Ainsi en jouant au « poupée russe » on peut imbriquer plusieurs panneaux (`TPanel`) ou autres composants (et jouer éventuellement avec les propriétés `BevelInner`, `BevelOuter`, `BevelWidth`[35], `BorderStyle` et `BorderWith` du panneau) afin de contrôler la présentation facilement.

[35] « *Bevel* », en anglais, désigne une biseau, c'est ce qui donne l'aspect 3D au panneau.

6.2.2 La séparation mobile

La classe `TSplitter` permet de gérer des séparations mobiles (que l'utilisateur pourra contrôler avec la souris) entre diverses partie d'une fenêtre.

Ainsi la fiche est conçue de la manière suivante :

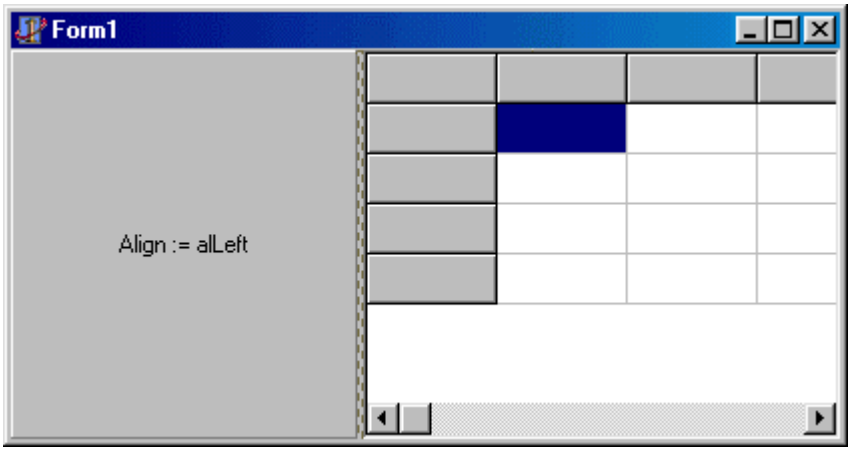

Figure 26. Le contrôle Splitter

A l'exécution, l'utilisateur pourra déplacer la frontière : Le curseur de la souris change de forme en arrivant sur le séparateur, l'utilisateur clique, déplace la frontière et relâche la souris :

Figure 27. L'utilisation du contrôle `Splitter`

Pour réaliser ce genre de comportement, procédez de la manière suivante :

- Placez un premier composant (par exemple un `TPanel`) à l'intérieur de la fiche ou du conteneur dans lequel vous voulez créer le comportement.
- Réglez sa propriété `Align` avec une des valeurs `alTop`, `alLeft`, `alRight` ou `alBottom`.
- Posez dans la fiche un objet `TSplitter`.
- Réglez la propriété `Align` du séparateur avec la même valeur.
- Posez un autre composant dans la fiche (dans notre exemple un tableau de chaînes de caractères).
- Réglez sa propriété `Align` à la valeur `alClient`.

Remarque

Vous pouvez en fait poser plusieurs composants associés à des `Splitters` par la propriété `Align`. On peut par exemple imaginer : un panneau et un Splitter alignés à gauche ; un panneau et un Splitter alignés à droite et finalement un panneau aligné au centre.

Pratique

Réalisez les exemples précédents.

Pratique

Réalisez l'exemple suivant :

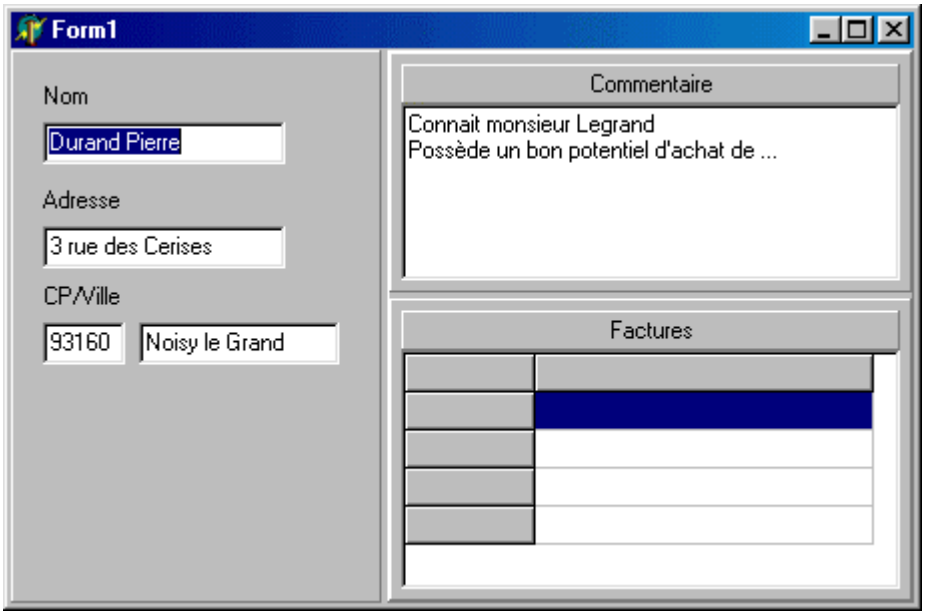

Figure 28. Exercice "Align" et "Splitter"

Voici quelques éléments pour réaliser cet exercice où nous avons deux objets `Splitter` :

- La partie gauche ne pose aucun problème avec un panneau contenant les composants d'édition.
- La partie droite part d'un panneau contenant lui-même deux autres panneaux et un splitter.
- Les titres sont réalisés à l'aide de panneau alignés en haut. Les autres composants utilisés sont un `TMemo` et un `TStringGrid`.

6.2.3 La propriété Anchor

Les contrôles possédant une propriété `Align` possèdent aussi une propriété `Anchor`. Celle-ci permet d'indiquer que le contrôle doit garder sa position par rapport à certains bords de son conteneur.

Ceci pourra être utile pour garder les boutons sur la partie gauche d'une boîte de dialogue dont la taille peut changer. Consulter l'aide de *Delphi* pour en savoir plus sur cette propriété `Anchor`.

6.3 LES EDITEURS

« *Editer* » est une mauvaise traduction du verbe anglais « *to edit* » qui signifie « préparer ou modifier un document avant son impression ».

Voici les principaux contrôles simples d'édition et leurs propriétés clef :

- `TEdit` : Zone de saisie générale permettant de manipuler un texte sur une ligne. Propriété clef :
  - `Text` : Contenu du texte disponible à l'écran.
- `TMaskEdit` : Zone de saisie générale ne permettant de manipuler un texte que si celui-ci obéit à une forme type (le masque). Propriétés clef :
  - `Text` : Contenu du texte disponible à l'écran.
  - `EditMask` : Masque de saisie permettant de préciser des règles quant à la place de certains caractères obligatoires, des règles de saisie en majuscules et en minuscules… Par exemple la saisie d'une date pourra être demandée avec « `!99/99/9999;1;_` ». Pour une description de toutes les possibilités, référez-vous à l'aide de *Delphi*.
- `TSpinEdit` : Contrôle de saisie de valeur numérique. Ce composant se présente comme un `TEdit` auquel sont accolées deux petites flèches pour augmenter ou diminuer la valeur affichée. Propriétés clef :
  - `Value` : La valeur en cours de saisie.
  - `MaxValue` et `MinValue` : Les valeurs extrêmes acceptables.
  - `Incrément` : La quantité dont augmente ou diminue la valeur lorsqu'on clique sur une flèche.

6.4 LES ACTIONS, LES MENUS ET LES BARRES D'OUTILS

Bien que cela ne soit pas toujours nécessaire, nous vous conseillons de toujours modéliser les actions de l'utilisateur dans un menu, une barre d'icônes… grâce à un composant `TAction` d'une `TActionList`.

6.4.1 Le programme d'exemple

Pour illustrer notre propos, nous allons écrire l'application suivante (fiche telle que vue depuis *Delphi*), qui permet de visualiser des images :

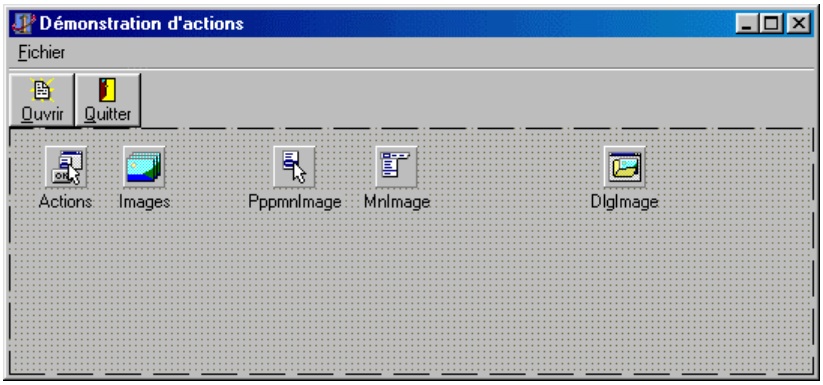

Figure 29. Visualisateur d'image : Programme d'exemple utilisant une liste d'actions

A l'utilisation cela pourra paraître comme cela, lorsqu'une image est chargée :

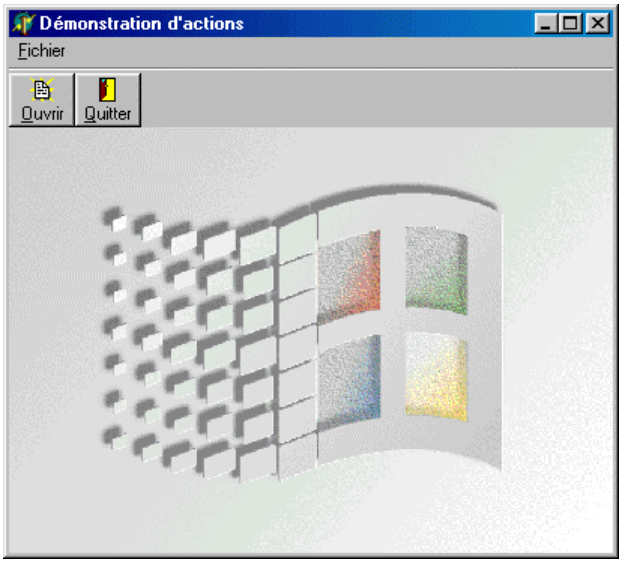

Figure 30. Programme d'exemple : Visualisateur d'image

Les caractéristiques de l'application sont les suivantes :

- Une barre d'outils contenant les icônes « *Ouvrir* » et « *Quitter* ».
- Un menu « Fichier » contenant les commandes « *Ouvrir* » et « *Quitter* ».
- Un menu contextuel (sur l'image) contenant les commandes « *Ouvrir* » et « *Quitter* ».

L'application est architecturée de la manière suivante :

- Un composant `Image` de la classe `TImage` destiné à la visualisation de l'image sélectionnée.
- Une boîte de dialogue `DlgImage` de la classe `TOpenPictureDialog` destinée à choisir un fichier graphique à charger dans le composant `Image`.
- Un menu principal `MnImage` de la classe `TMainMenu`.
- Un menu contextuel `PppmnImage` de la classe `TPopupMenu`. (La propriété `PopupMenu` de l'image fait référence à ce menu contextuel).
- Une barre d'outils de la classe `TToolBar`.
- Une liste d'actions `Actions` de la classe `TActionList`.
- Une liste d'images `Images` de la classe `TImageList`.

Pratique

Réaliser un programme qui obéisse à cette architecture d'application afin de pouvoir réaliser cet exemple.

6.4.2 La liste d'images

La liste d'images, `Images`, de la classe `TImageList`, va être utilisée afin d'associer une icône à chaque élément de menu, barre d'outils…

Pour éditer la liste d'images, double cliquez sur celle-ci. Une boîte de dialogue apparaît afin de manipuler cette liste d'images :

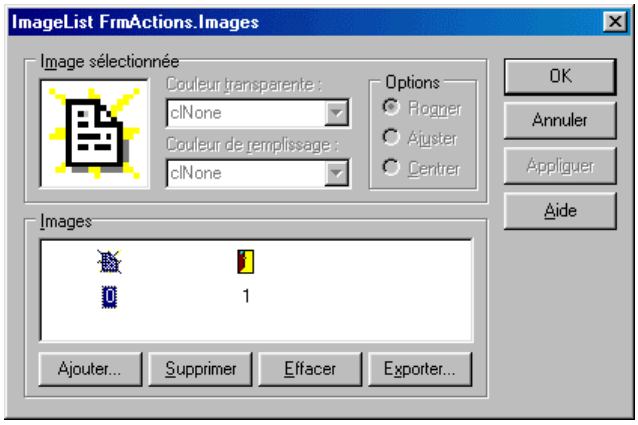

Figure 31. Les images d'un composant `TImageList`

En bas à gauche se trouve la représentation de toutes les images chargées dans la liste. Ces images sont numérotées à partir de 0. Quatre boutons permettent d'éditer cette liste. Une image est sélectionnée (en bleu), elle apparaît aussi en haut à gauche de la boîte de dialogue avec un facteur d'échelle supérieur — Certaines options sont disponibles dans la partie supérieure, consultez l'aide de *Delphi* à leur sujet.

Pour ajouter une icône à la liste, cliquez sur le bouton « Ajouter » :

- Une boîte de dialogue d'ouverture de fichier graphique apparaît.
- Sélectionnez et validez l'image souhaitée.

Remarque

Delphi est livré avec un bibliothèque d'images qui vous permettrons de satisfaire aux besoins les plus courants.

Cette bibliothèque se trouve dans le répertoire « `Program Files\Fichiers communs\Borland Shared\Images` ». Le sous répertoires « `Buttons` » contient des images bitmap permettant de concevoir la plupart des boutons et icônes.

Notez cependant que ces fichiers BMP contiennent deux images accolées dans un format de 32 pixels par 16 pixels : A gauche une icône normale, à droite une icône désactivée. Lorsque vous tentez d'importer une telle image dans une liste d'image, le composant cherchant à charger des images de 16 par 16 vous propose de séparer les deux images et de les charger séparément : répondez « Oui » à sa proposition, vous pourrez ensuite supprimer l'image inutile.

Pour supprimer simplement une image de la liste : sélectionnez-la (en cliquant dessus), puis appuyez sur la touche Suppr.

Pratique

Chargez la liste d'images avec 2 images, l'une pour la commande « *Ouvrir* », l'autre pour la commande « *Quitter* ».

6.4.3 La propriété Images

Plusieurs composants possèdent une propriété `Images` qui permet de leur associer une liste d'images. Ceci permettra d'associer une image de cette liste à certains composants qu'ils contiennent.

Pratique

Connecter les composants `Actions` (de la classe `TActionList`), `MnImage` (de la classe `TMainMenu`), `PppmnImage` (de la classe `TPopupMenu`) et la barre d'outils (de la classe `TToolbar`) à la liste d'images `Images`.

Pour ceci sélectionnez ces composants et affecter la valeur correcte à la propriété `Images` grâce à l'inspecteur d'objets.

6.4.4 La liste d'actions

Pour accéder à l'éditeur d'actions d'un composant `TActionList`, double cliquez sur celui-ci. Une fenêtre apparaît avec la liste des actions contenues dans ce composant. Par exemple :

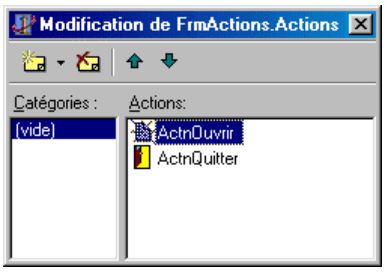

Figure 32. Les actions d'un composant `TActionList`

Cette fenêtre est organisée de la manière suivante :

- A droite se trouve la liste de tout ou partie des actions contenues dans la liste d'actions.
- Une action est sélectionnée (en bleu), elle est alors accessible depuis l'inspecteur d'objets.
- Les actions possèdent une propriété `Category`. Si celle-ci a été renseignée pour les actions, la partie gauche de la fenêtre montre toutes les catégories possibles. Lorsque vous sélectionnez une catégorie, vous ne voyez que les actions de cette catégorie (dans notre exemple, où il n'y a que 2 actions, nous n'avons pas utilisé la notion de catégorie).
- En haut de la fenêtre se trouve une barre d'outils avec 4 icônes :


Cette commande, aussi accessible par la touche Inser, permet d'ajouter une nouvelle action dans la liste.


Cette commande, aussi accessible par la touche Suppr, permet de supprimer l'action sélectionnée.


Cette commande permet de remonter l'action sélectionnée dans la liste.


Cette commande permet de descendre l'action sélectionnée dans la liste.

Le paramétrage d'une action

Lorsqu'une action est sélectionnée dans la liste, elle peut être manipulée depuis l'inspecteur d'objets. Voici les propriétés clefs :

- `Name` : Le nom du composant `TAction`.
- `Caption` : Le libellé associé à la commande que représente cette action.
- `ImageIndex` : Lorsqu'une liste d'images a été associée à la liste d'actions, `ImageIndex` permet de sélectionner une image dans cette liste grâce au numéro d'ordre de l'image.
- L'événement `OnExecute` : Pour écrire le gestionnaire d'événement correspondant à cette action. Notez que pour créer l'événement `OnExecute` associé à une action, vous pouvez double-cliquer sur cette action dans la liste.

Pratique

Nous allons concevoir deux actions, `ActnOuvrir` et `ActnQuitter`, et écrire les instructions suivantes dans leur gestionnaire d'événement `OnExecute` (double cliquez sur l'action pour créer l'événement) :

```
procedure TFrmActions.ActnOuvrirExecute(Sender: TObject);
begin
  if DlgImage.Execute then
    Image.Picture.LoadFromFile (DlgImage.FIleName);
end;

procedure TFrmActions.ActnQuitterExecute(Sender: TObject);
begin
  Close;
end;
```

L'instruction `DlgImage.Execute` ouvre la boîte de dialogue et renvoie `true` si l'utilisateur a sélectionné une image.

`DlgImage.FileName` donne le nom du fichier sélectionné.

Le composant `Image` (de la classe `TImage`) possède une propriété `Picture` qui permet de manipuler l'image en tant que telle. La méthode `LoadFromFile` permet de charger l'image depuis un fichier.

6.4.5 L'utilisation d'une action

Un certain nombre de composants sur lesquels l'utilisateur peut cliquer possèdent une propriété `Action` permettant de le lier à une action. Nous trouvons principalement :

- `TMenuItem`, item de menu contenu dans un objet de la classe `TMainMenu` ou `TPopupMenu`.
- `TButton`, `TBitBtn` et `TSpeedButton`, divers types de bouton.
- `TToolButton`, bouton contenu dans une barre d'outils, `TToolbar`.

Lorsqu'un de ces composants est connecté à une action il récupère automatiquement le paramétrage de l'action.

De plus lorsque l'action est modifié par programme, les éléments connectés réagissent en conséquence (par exemple si la propriété `Enabled` de l'action passe à `false`, les images sont automatiquement grisées).

Utilisation d'une barre d'outils

Pour créer un nouveau bouton, `TToolButton`, dans une barre d'outils, cliquez dessus la barre d'outils avec le bouton droit de la souris, et choisissez la commande « *Nouveau bouton* ».

Les propriétés `ShowCaption`, `Transparent`, `Flat`, permettent de contrôler l'apparence de la barre d'outils.

Pratique

Réalisez cet exemple.

Créez une nouvelle actions de style « *Aide* » ou « *A propos* » qui se contente d'afficher un message (fonction `ShowMessage`).

6.5 LES LISTES DE TEXTE

Plusieurs contrôles permettent de manipuler des listes de texte, par exemple pour sélectionner une valeur (Par exemple pour choisir entre « *Monsieur* », « *Madame* » ou « *Mademoiselle* »).

6.5.1 Le contrôle TListBox

Nous allons illustrer l'utilisation d'un composant `TListBox` grâce à l'application suivante :

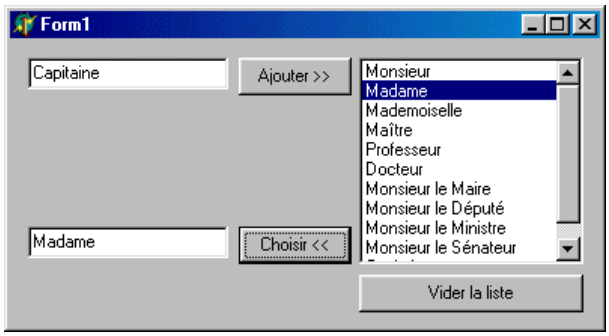

Figure 33. Programme d'exemple d'utilisation d'une `TListBox`

En haut à gauche se trouve un composant `TEdit` permettant de saisir du texte. Le bouton « *Ajouter* » permet d'ajouter le texte saisi dans la liste `TListBox` à droite.

Sous la liste se trouve un bouton « *Vider la liste* » permettant de vider la liste.

En bas à gauche se trouve un composant `TEdit` et un bouton « *Choisir* ». Lorsque l'on clique sur ce bouton, l'item sélectionné dans la liste (s'il existe) est transféré dans le composant `TEdit`.

Dans ce contexte, les deux propriétés majeures du composants TListBox sont :

- `Items` : Tableau dynamique de la classe `TStrings` permettant de manipuler plusieurs chaînes de caractères.
- `ItemIndex` : Valeur entière donnant la position de la chaîne sélectionnée dans le tableau (Attention le premier élément est à la position 0). Si aucun élément n'est sélectionné, `ItemIndex` vaut « -1 ».

Bien que nous ayons choisi de manipuler ces valeurs par le code, lors de l'exécution du programme, elles aurait pu être initialisées depuis l'inspecteur d'objet. Ainsi en sélectionnant la liste, vous voyez la propriété `Items` dans l'inspecteur d'objet. Double-cliquez sur le nom de cette propriété, une boîte de dialogue apparaît pour permettre de saisir tous les textes nécessaires.

La classe TStrings

Voici les principales propriétés et méthodes des objets de la classe `TStrings` (Rappel : la propriété `Items` d'une `TListBox` est de la classe `TStrings` — Au sein d'une `TListBox` ces propriétés et méthodes se manipulent avec l'objet `ListBox.Items`) :

- `Count` : Nombre de chaînes de caractères dans le tableau.
- `Strings` : Tableau de chaînes de caractères (les valeurs possibles de l'index vont de `0` à `Count-1`).
- `Add ('Text')` : Ajouter la chaîne de caractères passée en paramètre.
- `Clear` : Vider le tableau.
- `Delete (Indice)` : Retirer du tableau la chaîne d'indice donné.
- `LoadFromFile ('Fichier')` : Charger le tableau depuis un fichier.
- `SaveFromFile ('Fichier')` : Enregistre le tableau dans un fichier.

La propriété Strings

Lors de la conception d'une classe, les propriétés de type tableau peuvent être déclarée comme « propriété par défaut ». Ceci fait que le nom de la propriété est facultatif pour accéder au tableau. Ceci étant le cas pour la propriété `Strings` d'un `TStrings` on peut manipuler une chaîne de caractères d'un objet `ListBox` par une des deux syntaxes suivantes :

```
ListBox.Items.Strings [Indice]
```

ou

```
ListBox.Items [Indice]
```

Ceci nous amène tout naturellement au code de notre application d'exemple :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add (Edit1.Text);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  if ListBox1.ItemIndex >= 0 then
    Edit2.Text := ListBox1.Items [ListBox1.ItemIndex];
end;
```

Pratique

Réalisez l'exemple précédent.

Pratique

Réalisez l'exemple suivant :

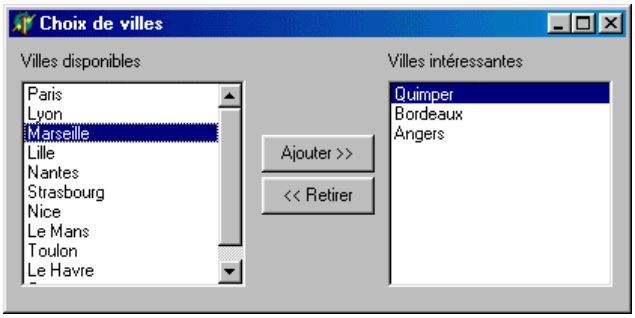

Figure 34. Pratique sur l'utilisation des `TListBox`

Nous avons deux listes : la première liste contient une série de villes disponibles (liste créée à la conception de la fiche), la deuxième contient des villes sélectionnées par l'utilisateur (liste vide à la conception).

Lorsque l'utilisateur clique sur le bouton « *Ajouter* » la ville sélectionnée dans la liste de gauche est mise dans la liste de droite, et est retirée de la liste de gauche. Le bouton « *Retirer* » provoque le déplacement inverse.

Lorsque l'utilisateur double clique sur une ville dans une liste, celle-ci passe automatiquement dans l'autre liste.

Remarque

Pour optimiser l'écriture de ce programme, vous pouvez écrire une procédure (sous la forme de méthode de classe) qui sert à transférer l'item courant d'une liste source vers une liste destination. La déclaration d'une telle procédure pourrait être :

```
type
  TForm1 = class(TForm)
    //…
  private
    class procedure MoveItem (Src, Dst : TListBox);
  public
    //…
  end;
```

Une fois cette procédure écrite, le déplacement de l'item courant d'une liste vers l'autre s'écrit simplement :

```
MoveItem(LbInteret, LbDispo);
```

6.5.2 Le contrôle TRadioGroup

Un contrôle `TRadioGroup` se gère exactement comme un `TListBox`.

Il possède quelques propriétés complémentaires permettant de gérer l'aspect du groupe de boutons :

- `Caption` : Pour afficher un titre dans le cadre englobant les boutons.
- `Columns` : Pour indiquer le nombre de colonnes afin de répartir les boutons.

Pratique

Réaliser le programme suivant qui fonctionne exactement comme celui « Figure 33. Programme d'exemple d'utilisation d'une `TListBox` ».

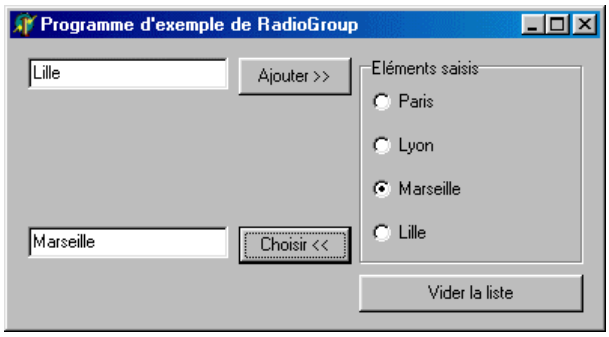

Figure 35. Programme d'exemple d'utilisation d'un `TRadioGRoup`

6.5.3 Le contrôle TMemo et TRichEdit

Le contrôle `TMemo` permet de manipuler un texte de plusieurs lignes, grâce à une propriété Lines qui est toujours de la classe `TStrings`.

Pratique

Réalisez le programme suivant qui permet de lire et d'enregistrer un fichier texte (utilisez les méthodes `Memo.Lines.LoadFromFile` et `Memo.Lines.SaveFromFile`). Vous aurez besoin de composants `TOpenDialog` et `TSaveDialog` pour demander les noms de fichier).

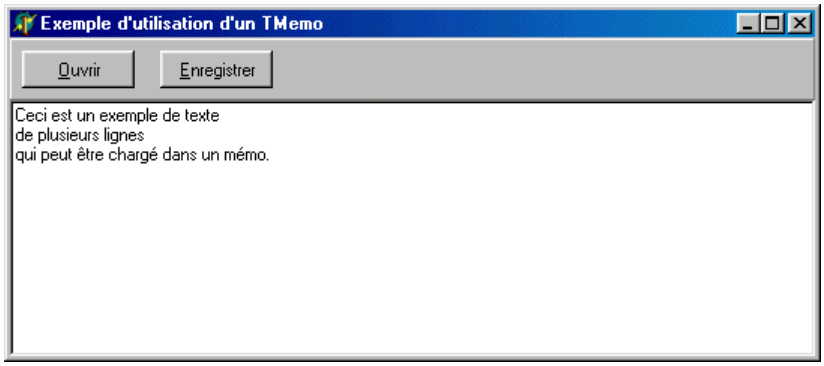

Figure 36. Programme d'exemple d'utilisation d'un `TMemo`

Le contrôle `TRichEdit` se comporte comme une version étendue du contrôle `TMemo` afin de pouvoir manipuler du texte enrichi (*RTF*). Ce contrôle possède aussi une propriété `Lines` de la classe `TStrings`. Il possède d'autres propriétés et méthodes permettant d'exploiter l'enrichissement du texte. Consultez l'aide de *Delphi* pour exploiter ce contrôle.

6.5.4 Le contrôle TCombBox

Un contrôle `TComboBox` se présente comme la combinaison d'un `TEdit` et d'un `TListBox`. A l'écran vous voyez un éditeur et à droite une flèche vers le bas permettant de faire apparaître la liste avec tous les choix proposés.

En tant que combinaison d'un `TEdit` et d'un `TListBox`, ce contrôle présente les propriétés clefs de chacun de ces composants :

- `Text` : Le texte saisi ou sélectionné dans le contrôle.
- `Items` : La liste des valeurs proposées. L'utilisateur aura accès à ces valeurs de la liste soit en cliquant sur la flèche à droite du contrôle, soit en utilisant les touches F4 ou Alt ↓, ou encore en se déplaçant dans les valeurs possibles grâce aux flèches ↓ ↑.
- `ItemIndex` : L'éventuel indice dans la liste de l'item sélectionné (voir la propriété `Style` ci-après).

Nous trouvons aussi une propriété complémentaire importante qui détermine comment utiliser le contrôle :

- `Style`. Les deux principales valeurs pour cette propriété sont les suivantes :
  - `csDropDown` : L'utilisateur peut saisir dans l'éditeur du texte qui n'appartient pas à la liste (dans ce cadre, la propriété `ItemIndex` n'a pas de sens).
  - `csDropDownList` : L'utilisateur doit sélectionner un item de la liste.

Pratique

6.6 LA CASE A COCHER ET LE BOUTON RADIO

6.7 LE DRAG AND DROP

6.8 LE TIMER

7. LA CLASSIFICATION DE L'INTERFACE

Remarque

Le contenu de cette section reprend le contenu d'un article de Jean-Paul Pruniaux publié dans la revue « Programmez » afin d'exposer certaines techniques de maintenance sous Delphi :

Delphi — Techniques de maintenance

La programmation orientée objet nous promet, entre autres, une maintenance plus facile de vos applications — Aussi, un véritable environnement orienté objets devrait pouvoir vous fournir des outils et des techniques allant dans ce sens. Nous allons examiner ici quelques cas de figure illustrant ces techniques.

Jean-Paul Pruniaux

Avant les techniques orientées objets, les meilleures techniques d'organisation du programme reposaient sur une bonne exploitation de la programmation structurée.

Un bon découpage en procédure vous donne des points bien précis dans le programme où un problème particulier est traité. Ainsi des routines données traitent de problèmes statistiques, d'autres de problèmes d'affichage graphique, ou d'autres encore des problèmes d'impression de fichiers…Ceci vous donne des espèces de points de passage obligés dans le programme, points où un problème particulier est traité. Si une erreur est commise dans le traitement du problème, la maintenance du programme est facilitée, car il ne s'agira que d'une procédure à corriger. Une fois celle-ci corrigée, une centaine d'erreurs peuvent disparaître d'un seul coup.

7.1.1 L'ancêtre — Un centralisateur de solutions

Comment alors, obtenir le même genre de centralisation d'un problème avec la programmation orientée objet?

Nous allons bien sûr utiliser des notions fondamentales de l'objet: "Héritage", "Polymorphisme" et "Abstraction". Nous allons voir comment au travers d'un exemple concret.

Prenons une application gérant un fichier de clients, un fichier de fournisseurs et un fichier de commandes. Nous pouvons avoir 3 fiches, l'une dédiée à la consultation ou la saisie d'un client, l'une à un fournisseur et une dernière à une commande.

Ces 3 fiches, bien que manipulant des fichiers différents possèdent des caractéristiques communes: une table à éditer, un bouton "Modifier", un bouton "Nouveau", et une fois en mode édition un bouton "Sauver" et un bouton "Annuler", et peut être d'autres caractéristiques.

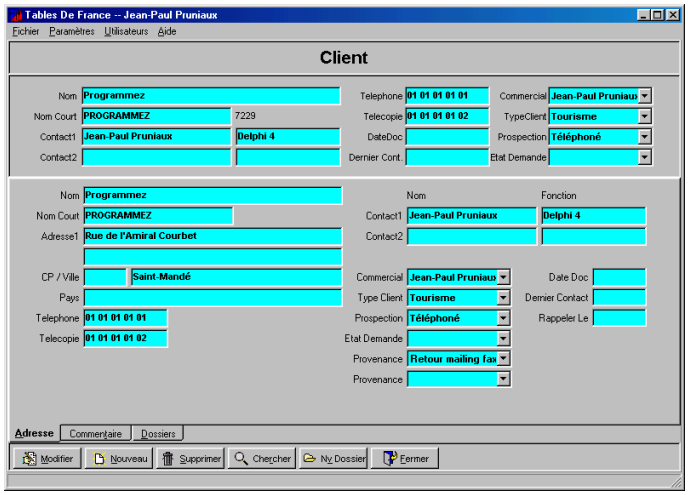

Figure 37. Exemple de fiche "Client" possédant une gestion semblable à une fiche "Fournisseur"

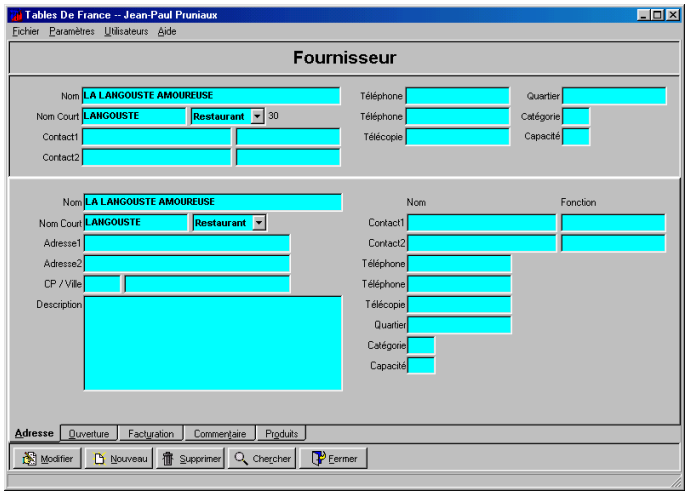

Figure 38. Exemple de fiche "Fournisseur" possédant une gestion semblable à une fiche "Client"

Pour centraliser les caractéristiques communes à ces diverses fiches, nous allons créer une nouvelle classe de fiches (et donc un nouveau type de données) que nous pourrions appeler une "fiche de saisie". Celle-ci pourrait ressembler à la figure 3.

Cette fiche n'est pas destinée à être utilisée telle quelle mais à servir d'ancêtre commun aux fiches de saisie — C'est ce que nous appelons une classe abstraite, car cette fiche n'apparaîtra jamais telle quelle dans le programme. En français, dans la notion d'abstrait nous retrouvons des choses comme "dégager une idée" et " sans existence réelle"; et nous avons effectivement dégager l'idée "saisie de données".

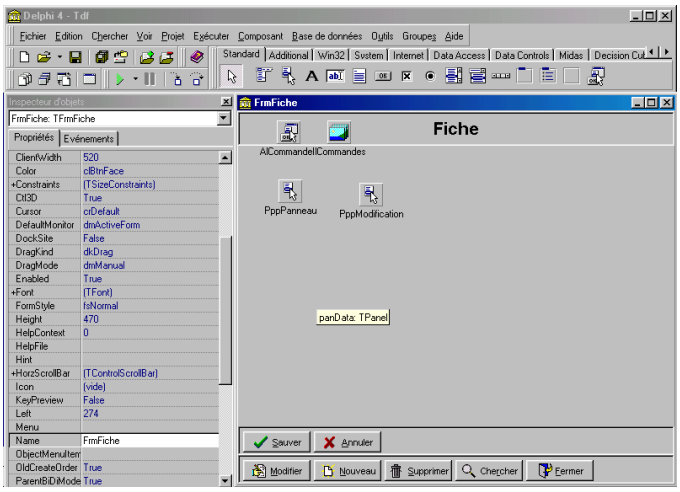

Figure 39. Exemple d'ancêtre commun aux fiches de saisie.

La première précaution à prendre consiste donc, après avoir conçu une telle fiche, à s'assurer que Delphi ne créera pas automatiquement une telle fiche. Pour ceci, utilisez la commande "Projet/Options/Fiches" et retirez cette nouvelle fiche de la liste des "fiches créées automatiquement".

7.1.2 Rattraper une erreur de conception

La séquence correcte de travail consiste à créer en premier lieu les fiches abstraites, puis à créer de nouvelles fiches à partir de ces fiches abstraites.

Pour créer la fiche de saisie de clients en respectant cette séquence, vous utiliserez la commande "Fichier/Nouveau…" — Une boîte de dialogue apparaît, allez alors dans l'onglet correspondant aux composants de votre application, choisissez l'ancêtre à partir duquel créer la nouvelle fiche et assurez-vous que le bouton radio "Hériter" soit coché (Voir la figure 4).

Malheureusement, il arrive que dans le processus de développement d'un programme on ne voie pas toujours du premier coup comment gérer les héritages. Aussi, peut-il arriver que nous commencions à concevoir une fiche "Client", puis une fiche "Fournisseur" et que là, nous réalisions qu'une classe abstraite serait la bienvenue pour gérer les caractéristiques communes entre ces deux classes de fiches.

La solution consiste alors à créer la classe abstraite que nous jugeons nécessaire puis à corriger le diagramme d'héritage pour demander aux fiches "Client" et aux fiches "Fournisseur" d'hériter des fiches de saisie.

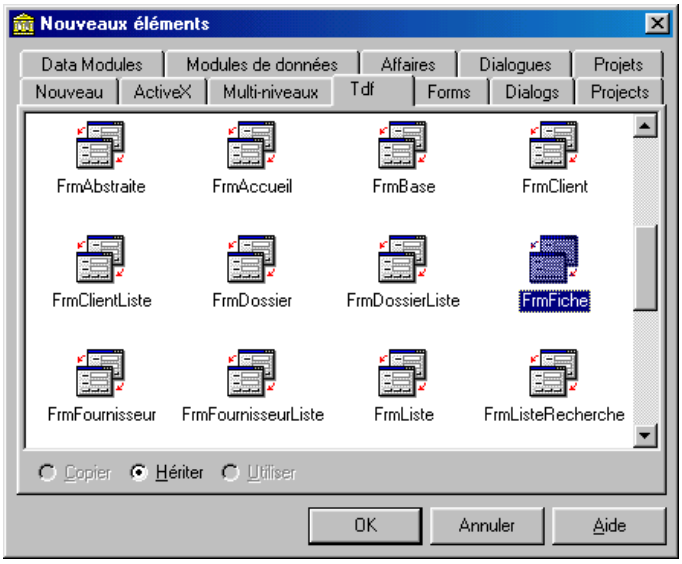

Figure 40. Création d'une nouvelle fiche héritant des caractéristiques d'une fiche existante

Les étapes sont alors les suivantes:

Pour vous simplifier la vie, commencez par vérifier qu'aucun composant de la fiche client ne porte de nom identique avec les composants de la fiche de saisie. Si tel était le cas, renommez ces composants soit dans la fiche de saisie soit dans le fiche client (meilleur choix).

Nous allons maintenant commencer à corriger le diagramme d'héritage: Allez dans le source de la classe des fiches clients et localisez les lignes :

```
type
  TFrmClient = class (TForm)
```

Ceci signifiant que les fiches clients héritent des fiches vierges de la bibliothèque de Delphi.

Corrigez cette déclaration en

```
type
  TFrmClient = class (TFrmFiche)
```

Où "TFrmFiche" est le nom que vous avez donné à la classe abstraite gérant les caractéristiques des fiches de saisie.

Maintenant ceci ne suffit pas pour que Delphi prenne complètement en compte le changement du diagramme d'héritage.

Nous avons en effet partiellement corrigé le source Pascal "fClient.pas" dans le cadre de notre exemple, mais nous n'avons pas corrigé le fichier contenant les valeurs initiales à affecter aux fiches clients, "fClient.dfm" dans le cadre de notre exemple — Il va falloir ici préciser qu'il faut lire aussi les valeurs initiales contenue dans "fFiche.dfm".

Pour ceci, commencez par afficher la fiche client à l'écran (le dessin de la fiche, pas le contenu du source Pascal), puis appuyez sur les touches <Alt>+<F12>. Le mode de l'environnement de travail change pour vous montrer le contenu du fichier "fClient.dfm". Vous obtenez dans l'éditeur de texte quelque chose comme:

```
object FrmClient: TFrmClient
  Left = 29
  Top = 27
  Width = 747
  Height = 540
  Caption = 'FrmClient'
  PixelsPerInch = 96
  TextHeight = 13
```

Vous retrouvez dans ce source, l'ensemble des valeurs que vous examineriez avec l'inspecteur d'objet alors que vous êtes dans la fiche client ou un de ses composants.

Corrigez la première ligne de ce source, en remplaçant le mot "object" par le mot "inherited":

```
inherited FrmClient: TFrmClient
  Left = 29
  Top = 27
  Width = 747
  Height = 540
  Caption = 'FrmClient'
  PixelsPerInch = 96
  TextHeight = 13
```

Maintenant les fiches de la classe clients "sauront" qu'elles devront aussi lire leur valeur par défaut dans la classe des fiches de saisie.

Appuyez sur les touches <Alt>+<F12> pour visualiser de nouveau la fiche client, et sauvegardez cette fiche. Lors du processus de sauvegarde, Delphi analyse toutes les déclarations et leur cohérence entre le contenu du fichier "*.dfm" et le fichier "*.pas" — La liste des unités utilisées est aussi mise à jour dans le source Pascal.

Le diagramme d'héritage a été corrigé pour la fiche client. Vous devriez voir maintenant dans la fiche client tout ce qui a été conçu dans la fiche de saisie et tout ce qui existait déjà dans la fiche client. Vous allez devoir donc faire le ménage dans la fiche "Client" pour retirer ce dont vous n'avez plus besoin. Notez au passage que ce qui a été déclaré dans la fiche ancêtre ne peut être retiré de la fiche "Client".

Reprenons ce scénario pour corriger la fiche "Fournisseur":

- Renommez tous les composants de la fiche fournisseur possédant un nom identique à ceux de la fiche de saisie.
- Dans le source "fFournisseur.pas", corrigez la déclaration "TFrmFournisseur = class (TForm) en "TFrmFournisseur = class (TFrmFiche)".
- Appuyez sur <F12> pour voir le dessin de la fiche "Fournisseur".
- Appuyez sur <Alt>+<F12> pour voir le contenu du fichier "fFournisseur.dfm".
- Corrigez la première ligne "object FrmFournisseur : TFrmFournisseur" en "inherited FrmFournisseur : TFrmFournisseur".
- Appuyez de nouveau sur les touches <Alt>+<F12> pour voir la fiche fournisseur.
- Sauvegardez cette fiche.
- Réorganisez votre nouvelle fiche fournisseur.

7.1.3 Polymorphisme

"Polymorphisme", un mot grec signifiant "Capacité d'assumer plusieurs formes", tel ce caméléon étudié en classe de sixième.

En fait dans notre exemple précédent, nous avons déjà vu le polymorphisme à l'œuvre. Note fiche de saisie "TFrmFiche" peut assumer la forme "TFrmClient" ou la forme "TFrmFournisseur". Et si nous modifions la fiche de base, "TFrmFiche", les fiches descendantes vont hériter de ces modifications. Ajoutons par exemple un logo en haut à gauche de la fiche de base, ce logo sera aussi présent dans les fiches "Client" et "Fournisseur". Déplaçons ce logo dans la fiche client, et il occupera une nouvelle position dans cette fiche: la forme de la fiche de saisie a de nouveau changé dans le contexte "Client".

Comment revenir à la forme de l'ancêtre? Comment remettre le logo de la fiche "Client" à la place qu'il occupait dans la fiche de saisie? En cliquant sur le bouton droit de la souris, soit sur l'objet concerné, soit dans la propriété adéquate de l'inspecteur d'objet, vous trouvez dans le menu contextuel la commande suivante: "Revenir à hériter". En choisissant cette option, vous ramenez l'objet sélectionné ou cette propriété dans le contexte qu'il avait dans la fiche de base.

Mais là ne s'arrête pas le polymorphisme. Revenons à ce parallèle que nous évoquions au début de cet article dans un contexte structuré: "le découpage en procédures pour centraliser les traitements communs".

Imaginions que dans la fiche de saisie nous avions une méthode telle que:

```
procedure TFrmTable.AspectModifier(Modif: boolean);
begin
if Modif then
  begin
  PanMoved.PopupMenu      := PppModification;
  PanButton.Visible       := false;
  PnlModification.Visible := true;
  end
else
  begin
  PanMoved.PopupMenu      := PppPanneau;
  PnlModification.Visible := false;
  PanButton.Visible       := true;
  end
end;
```

Cette méthode est destinée à gérer l'aspect de la fiche afin de montrer ou cacher des choses ou afin de changer l'apparence de la fiche en fonction de son contexte d'édition. Si nous sommes en train de modifier des données, le panneau contenant les boutons "Sauver" et "Annuler" doit être visible, sinon il doit être caché. De la même manière, nous pouvons convenir que lors d'une modification les champs à saisir sont d'une couleur alors qu'en consultation ils sont d'une autre couleur.

Ce que doit faire cette procédure doit être personnalisé dans les sous-classes "TfrmClient", "TFrmFournisseur", etc…

Voici comment dot être déclarée cette méthode dans la fiche de saisie:

```
TFrmFiche = class(TForm)
    //...
  private
    { Déclarations privées }
    //...
  protected
    //...
    procedure AspectModifier (Modif : boolean);
              virtual;
    //...
public
    { Déclarations publiques }
    //...
  end;
```

Elle est déclarée dans une section "protégée" (protected) car seuls les membres de cette classe et de ses sous-classes pourront y accéder.

Elle est déclarée "virtuelle" (virtual) car nous désirons la personnaliser dans les sous-classes: Nous désirons introduire du polymorphisme sur cette méthode. Cette méthode est virtuelle car elle est incomplète et elle devra ou pourra être personnalisée dans une sous-classe — Ceci rejoint l'idée décrite par le terme "virtuelle": "Qui n'a pas d'existence réelle, mais qui se comporte comme si elle existait réellement". En fait bien qu'elle ne soit pas définie complètement (elle ne gère pas le changement de couleur des champs d'édition de la fiche client et de la fiche de fournisseur), elle existe néanmoins et peut être manipulée depuis la fiche de saisie.

Dans une sous classe, cette méthode pourra revêtir l'état suivant:

```
  TFrmClient = class(TFrmFiche)
    //...
  private
    { Déclarations privées }
  protected
    //...
    procedure AspectModifier (Modif : boolean);
              override;
    //...
  public
    { Déclarations publiques }
    //...
  end;

procedure TFrmClient.AspectModifier(Modif: boolean);
begin
  inherited AspectModifier(Modif);
  if Modif then
    begin
    dbeAdNomCourt.Color := clWindow;
    dbeAdNomCourt.Color.ReadOnly := false;
    //...
    end
  else
    begin
    dbeAdNomCourt.Color := clAqua;
    dbeAdNomCourt.Color.ReadOnly := true;
    //...
    end
end;
```

Dans la sous-classes elle est déclarée "surchargée" (overload) pour indiquer que nous introduisons ici une personnalisation à la méthode virtuelle de la classe ancêtre. Cette personnalisation peut intervenir, avant, après ou à la place de l'instruction " inherited AspectModifier(Modif);" qui appelle la méthode définie dans la classe ancêtre.

Le polymorphisme intervient aussi dans la gestion des événements, sous un aspect légèrement différent. Delphi créée des méthodes particulières pour gérer les événements. Nous pouvons trouver par exemple dans la classe ancêtre un gestionnaire d'événements tel que:

```
  TFrmFiche = class (TForm)
    procedure ActnModifierExecute(Sender: TObject);
    procedure ActnNouveauExecute(Sender: TObject);
    //...
  end;

procedure TFrmFiche.ActnModifierExecute
(Sender: TObject);
begin
  TblDonnees.Edit;
  AspectModifier (true);
end;

procedure TFrmFiche.ActnNouveauExecute
(Sender: TObject);
begin
  TblDonnees.Insert;
  AspectModifier (true);
end;
```

Ces gestionnaires d'événements utilisent la méthode "AspectModifier" précédemment écrite, et forcent le contexte d'édition ou de modification, imaginons par exemple que nous avons un objet "TTable", TblDonnees, permettant de manipuler un fichier de données.

Si dans une des sous-classes nous définissons un nouveau gestionnaire d'événement pour personnaliser la gestion de l'événement de début de modification nous pourrons avoir quelque chose du genre:

```
  TFrmFournisseur = class (TFrmFiche)
    procedure ActnModifierExecute(Sender: TObject);
    procedure ActnNouveauExecute(Sender: TObject);
    //...
  end;

procedure TFrmFournisseur.ActnModifierExecute
(Sender: TObject);
begin
  inherited;
  TblProduits.Edit;
end;

procedure TFrmFournisseur.ActnNouveauExecute
(Sender: TObject);
begin
  inherited;
  TblProduits.Insert
end;
```

Dans la fiche de fournisseurs, l'objet "TblDonnees", sert à manipuler les données d'un fournisseur, mais nous devons aussi gérer un autre objet "TTable", TblProduit" pour manipuler les produits de ce fournisseur.

Ici nous avons affaire au mot clef "inherited" seul. Ceci demande le traitement de l'événement défini dans la classe ancêtre. Il ne s'agit pas de surcharge de méthode mais de surcharge de gestionnaire d'événement. La différence réside dans la manière dont la méthode a été appelée. Dans le premier cas, "AspectModifier" est une méthode qui a été appelée explicitement par une instruction pascal (AspectModifier (true);). C'est le programmeur qui a écrit le code déclenchant l'appel de la méthode. Dans le deuxième cas, " ActnModifierExecute" ou " ActnNouveauExecute", c'est le

mécanisme de gestion des événements décrit dans la bibliothèque de Delphi qui appelle le code concerné. Ce mécanisme de gestion d'événements utilise d'autres types de méthode que les méthodes virtuelles (elles ne sont pas déclarées avec l'attribut "virtual" — La description de cet autre mécanisme sort du thème de cet article).

Voici quelques outils qui permettent donc de gérer les différences et les ressemblances entre les fiches, et qui permettent de personnaliser le comportement d'une fiche par rapport à une autre. Ceci devrait déjà vous aider à optimiser votre code.

Voyons maintenant un autre type de problème.

7.1.4 La modification du comportement d'un composant de la VCL

Vous venez d'écrire une application, et lors de la phase de tests, vous vous rendez compte que le comportement d'un des composants utilisés doit être modifié.

Voici par exemple deux cas concrets qui sont arrivé à l'auteur de cet article.

Une application utilise des objets "TTable" pour manipuler des tables Paradox, et le client se plaint que régulièrement au démarrage de l'application il obtient des messages "Error index out of date" et l'application refuse de démarrer. La solution de dépannage est simple, il faut reconstruire les index. Mais en parallèle une enquête doit être menée pour savoir ce qui casse ces index afin de trouver une solution et palier à cette perte d'intégrité dans la base de données.

Après quelques consultations appropriées dans les forums de Compuserve il ressort que cette erreur peut intervenir dans les circonstances suivantes:

- Défaillance dans le réseau (ce qui n'était pas le cas).
- Mauvaise utilisation du programme: certains utilisateurs ferment leur ordinateur brutalement sans terminer le programme (ce qui était et est toujours le cas.

Cette même enquête offre des solutions pour lutter contre le problème. Entre autres, palier à une faiblesse des composants TTable de la VCL en forçant l'écriture des données sur le disque ou sur le réseau juste après la modification d'un enregistrement.

Autre cas concret: une application utilise des objets "TpageControl" et "TTabSheet" dans ses écrans, et le client demande à ce que le titre de la page active soit écrit en gras afin que nous sachions facilement quelle est la page active. Aucune propriété standard des composants concernés ne semble résoudre le problème, il semble qu'il faille gérer pour chaque composant un dessin personnalisé.

Dans l'un ou l'autre cas, pour quelqu'un ayant une faible maîtrise de l'objet, il semble qu'il faille intercepter les événements adéquats et écrire pour chaque objet un gestionnaire d'événement. Ainsi l'application comporte une centaine d'objet "TTable" et nous pensons à intercepter une centaine d'événements "BeforePost" afin d'écrire une centaine de fois le code forçant l'écriture de données sur le réseau. Ou bien encore l'application comporte une dizaine d'objets "TPageControl" et nous devons intercepter une dizaine de fois l'événement "OnDrawTab" afin de personnaliser le dessin des onglets.

Fort heureusement, il existe une autre solution, purement objet, qui consiste à modifier le comportement des objets "TTable" ou "TPageControl". Prenons par exemple la manière dont a été traitée la modification de comportement de l'objet "TPageControl" — Un nouveau composant "TJppPageControl" a été conçu pour résoudre le problème de l'affichage des onglets (voyez notre précédent article sur l'écriture de composants avec Delphi pour étudier le listing ci-après et pour installer le composant):

```
unit JppPageControl;

interface

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

type
  TJppPageControl = class(TPageControl)
  protected
    procedure DrawTab(TabIndex: Integer; const Rect: TRect; Active:
Boolean); override;
  public
    constructor Create (AOwner : TComponent); override;
  published
    { Déclarations publiées}
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Jpp InOut', [TJppPageControl]);
end;

{ TJppPageControl }

constructor TJppPageControl.Create(AOwner: TComponent);
begin
  inherited Create (AOwner);
  Font.Style := [fsBold];
  OwnerDraw := true;
end;

procedure TJppPageControl.DrawTab(TabIndex: Integer; const Rect: TRect;
  Active: Boolean);

  var
    Text : string;
    Pos  : integer;
    xPos : integer;

  procedure RemoveAmpersand (Txt : string);
    var
      i : integer;
      n : integer;
    begin
    Pos := 0;
    n := Length (Txt);
    Text := '';
    for i := 1 to n do
      if Txt [i] <> '&' then
        begin
        Text := Text + Txt [i];
        end
      else
        Pos := Length (Text)+1;
    if Pos >= Length (Text) then
      Pos := Length (Text) - 1;
    end;
```

```
  begin
  Canvas.FillRect(Rect);
  Canvas.Font.Assign (Font);
  if Active then
    Canvas.Font.Style := [fsBold]
  else
    Canvas.Font.Style := [];
  RemoveAmpersand (Tabs[TabIndex]);
  Canvas.TextOut (
    Rect.Left + (Rect.Right-Rect.Left-Canvas.TextWidth (Text)+1) div 2,
    Rect.Top  + (Rect.Bottom-Rect.Top-Canvas.TextHeight(Text)+1) div 2 -1,
    Text);
  if Pos > 0 then
    begin
    xPos := Canvas.TextWidth (Copy (Text, 1, Pos-1));
    Canvas.Font.Style := Canvas.Font.Style + [fsUnderline];
    Canvas.TextOut (
      Rect.Left + (Rect.Right-Rect.Left-Canvas.TextWidth (Text)+1) div 2 +
xPos,
      Rect.Top  + (Rect.Bottom-Rect.Top-Canvas.TextHeight(Text)+1) div 2 -1,
      Text[Pos]);
    end;
  end;

end.
```

Le problème devient alors: "Comment remplacer le composant "TPageControl" par le composant "TJppPageControl" dans les fiches de l'application?"

Une fois que le nouveau contrôle a été écrit et testé, la procédure devient la suivante (elle ressemble à ce que nous avons déjà vu pour corriger un diagramme d'héritage de fiche):

- Charger une fiche utilisant des "TPageControl".
- Appuyez sur les touches <Alt>+<F12> afin de voir le contenu du fichier DFM.
- Cherchez les textes "TPageControl" et remplacez-les par des textes "TJppPageControl".
- Revenez en mode fiche en appuyant à nouveau sur les touches <Alt>+<F12>.
- Enregistrez la fiche. Lors de l'enregistrement Delphi analyse la cohérence entre le fichier DFM et le source PAS. Il détecte donc que certains composants ne sont plus du type attendu et demande pour chacun d'eux s'il doit corriger la déclaration "??? : TPageControl" en "??? : TJppPageControl" — répondez "oui" à chaque fois.

Nous avons ainsi corrigé le problème sur certains composants de la VCL, et il va sans dire que si nous avons d'autres problèmes à corriger sur ces composants, cela sera encore plus simple, car maintenant notre application utilise des composants dont nous avons la maîtrise des sources.

Voici quelques conseils et quelques techniques qui devraient vous aider à tirer davantage de bénéfice de la POO, maintenant à vous de les exploiter.

8. ANNEXES

8.1 LES FONCTIONS DE CONVERSION

Le *Pascal Objet* de *Delphi* permet de manipuler un certain nombre de types de données de base donc voici les plus importants :

- `Integer` : Valeur entière sur 32 bits.
- `String` : Chaîne de caractères.
- `Double` : Valeur numérique en virgule flottante.
- `TDateTime` : Valeur temporelle (date et heure).

L'unité `SysUtils` intègre un certain nombre de fonction permettant de convertir les valeurs d'un type en un autre :

- `IntToStr` :
- `StrToInt` :
- `DateTimeToStr` :
- `StrToDateTime` :
- `DateToStr` :
- `StrToDate` :
- `TimeToStr` :
- `StrToTime` :
- `FloatToStr` :
- `FloatToStrF` :
- `StrToFloat` :

9. GLOSSAIRE

Composant Objet manipulable dans l'environnement de travail graphique de *Delphi*. Cet objet ne sera pas nécessairement vu par l'utilisateur du programme.

Conteneur Un objet (un composant ou un contrôle) peut en contenir d'autres objets (composants ou contrôles). Les relations Parent-Enfant (voir Parent) ou Propriétaire-Objet (voir Propriétaire) sont des relations de contenance. Par exemple, une fiche contient un certain nombre de composants.

Contrôle Objet que l'utilisateur verra à l'exécution du programme

Délégation d'événement Mécanisme par lequel un objet qui reçoit un événement peut rediriger cet événement vers un autre objet qui en réalisera le traitement.

Evénement « *Tout ce qui arrive, tout fait qui intervient dans une continuité et en caractérise un moment…* » (Dictionnaire Logos de Bordas).

On pourrait résumer un événement à « *Il se passe quelque chose* ».

Flux Mécanisme générique d'entrée sortie (Flux de données).

Message 1/ (Au sens Windows) Evénement codifié généré par le système, structure de données envoyée par le système pour décrire cet événement.

2/ (Au sens POO) Le fait de solliciter un objet ou une classe grâce à l'opérateur point, « . ».

Parent Contrôle dans lequel un autre contrôle est affiché (Notion issue de l'organisation interne de *Windows*). Attention, le parent d'un contrôle n'est pas toujours la fenêtre dans lequel il est affiché. Prenons l'exemple d'une fenêtre contenant un panneau, ce panneau contenant des zones d'édition : le parent des zones d'édition est le panneau, le parent du panneau est la fenêtre.

Propriétaire (Owner) Lorsque le propriétaire d'un composant (ou d'un objet) est détruit, ce composant (ou cet objet) est lui aussi détruit. Comme le composant (ou l'objet) appartient à son propriétaire, il disparaît en même temps que son propriétaire.

10. INDEX

R

S

T

U

V

W