Introduction à C++ Builder

OURS BLANC DES CARPATHES ™

ISIMA 1988-1999

Table des matières

Table des illustrations

Figures

Programmes

Tableaux

1. C++ Builder : un environnement RAD basé sur C++

C++ Builder est le nouvel environnement de développement basé sur C++ proposé par Borland … pardon Inprise ! Fort du succès de Delphi, Borland a repris la philosophie, l'interface et la bibliothèque de composants visuels de ce dernier pour l'adapter depuis le langage Pascal Orienté Objet vers C++ répondant ainsi à une large faction de programmeurs peu enclins à l'utilisation du Pascal qu'ils jugent quelque peu dépassé.

1.1 Un environnement RAD

1.1.1 Philosophie

Tout d'abord C++ est un outil RAD, c'est à dire tourné vers le développement rapide d'applications (Rapid Application Development) sous Windows. En un mot, C++ Builder permet de réaliser de façon très simple l'interface des applications et de relier aisément le code utilisateur aux événements Windows, quelle que soit leur origine (souris, clavier, événement système, etc.)

Pour ce faire, C++ Builder repose sur un ensemble très complet de *composants visuels* prêts à l'emploi. La quasi totalité des contrôles de Windows (boutons, boîtes de saisies, listes déroulantes, menus et autres barres d'outils) y sont représentés, regroupés par famille. Leurs caractéristiques sont éditables directement dans une fenêtre spéciale intitulée *éditeur d'objets*. L'autre volet de cette même fenêtre permet d'associer du code au contrôle sélectionné.

Il est possible d'ajouter à l'environnement de base des composants fournis par des sociétés tierces et même d'en créer soit même.

Un outil RAD c'est également un ensemble de *squelettes* de projets qui permettent de créer plus facilement une application SDI ou MDI, une DLL, des objets OLE, etc. A chacun de ces squelettes est habituellement associé un *expert* qui par une série de boîtes de dialogues permet de fixer une partie des options essentielles à la réalisation du projet associé.

1.1.2 Limitations

Tout d'abord, il faut savoir que la technologie RAD ne s'applique qu'au squelette ou à l'interface d'une application. Bien entendu, toute la partie spécifique à votre projet reste à votre charge.

Du point de vue portabilité, le code C++ Builder n'est pas compatible C++ ANSI. Ceci est du à la gestion de la bibliothèque des composants visuels et en particulier de leurs *propriétés*. Pour l'heure, il suffit de savoir qu'une propriété d'un objet est assimilable à un attribut auquel on accède par affectation directe dans le code utilisateur. Toutefois, cet accès apparemment direct masque l'utilisation de méthodes d'accès en lecture et / ou écriture. Ce système de propriété a été mis au point par

Microsoft dans le cadre d'OLE puis démocratisé par le langage Visual Basic. Bien intégré dans le langage Pascal Orienté Objet, support, rappelons-le, de Delphi, il a fallu étendre le C++ pour l'y intégrer, notamment en ajoutant le mot clef non ANSI **__property**. De fait, tout le code C++ Builder qui fait appel à la VCL est non portable. En revanche, toute partie du code de l'application non directement liée à l'interface peut très bien être écrit en C++ ANSI. Plus que jamais, il convient d'adopter la décomposition Interface / Données / Méthodes pour l'écriture d'une application C++ Builder.

1.2 C++ Builder vis à vis de la concurrence

Loin d'être exhaustive, cette partie ne vise qu'à placer C++ Builder dans le cadre des outils RAD les plus présents sur le marché. Ayant peu travaillé avec Visual C++, je préfère ne pas présenter de comparatif avec cet outil par soucis d'équité. Je préciserai même, au bénéfice de Visual C++, que ce dernier accorde une bien meilleure place au modèle Document/Interface/Contrôleur que son rival, au détriment d'une interface un peu plus complexe à utiliser au premier abord.

1.2.1 C++ Builder vs Delphi... où les frères ennemis !

Tout d'abord, il faut savoir que, pour un même problème, l'exécutable fourni par C++ Builder est toujours un peu plus gros que celui issu de Delphi. Ceci tient au fait que la bibliothèque de composants visuels utilisée par les deux reste nativement celle de Delphi, aussi l'exécutable C++ Builder contient-il un module supplémentaire destiné à faire le lien avec des objets au format Delphi.

En dehors de ça, les deux outils sont très similaires. Mon choix se portera tout de même sur C++ Builder car ce dernier permet une meilleure intégration des contrôles ActiveX et autres mécanismes issus d'OLE.

1.2.2 Différences par rapport à Borland C++

Les différences par rapport à Borland C++ sont assez nombreuses. La première réside dans la nature du code produit. Si OWL, la bibliothèque de gestion de la programmation sous Windows et Borland C++ était 100% compatible C++ ANSI, la gestion de la VCL ne l'est pas pour les raisons exprimées au paragraphe précédent.

En outre, C++ Builder pose les problèmes communément liés aux outils de haut niveau. Par exemple, il est très difficile d'accéder directement aux messages Windows. En effet, s'il est toujours possible d'utiliser les primitives de l'API Windows, ces dernières ont elles-même été encapsulées dans une API de plus haut niveau, fournissant certaines valeurs par défaut à des paramètres clef.

En outre, certains messages Windows se révèlent totalement inaccessibles. Donnons un exemple lié aux boîtes d'édition de texte. Dans OWL, rappelons le, il était possible de passer la taille du tampon de saisie au constructeur de l'objet associé à une boîte d'édition. Il était alors possible d'associer un événement au remplissage du tampon de saisie. Avec C++ Builder, cette facilité a disparu. En outre, les événements liés aux groupes de boutons radio ou de cases à cocher sont à gérer individuellement pour chacun d'eux. Il n'est pas possible d'utiliser un événement de type

CHILD_NOTIFY autorisant la gestion globale d'un groupe de boutons par le contrôle groupe les englobant.

La gestion du modèle document / visualisation tant prônée par OWL (et reprenant les principes du découpage données / méthodes / interface) est ici abandonnée. De fait, on ne retrouve plus les fameuses classes TDocument et TView à la base d'un tel mécanisme. Il vous faudra composer vous même pour respecter un tel modèle.

Alors, à la question, faut il jeter Borland C++ au profit de C++ Builder ? je répondrai : faut voir … Si vous générez des applications de bases de données axées interface utilisateur la réponse est sans aucun doute : oui. A l'évidence, la réponse sera toujours oui … sauf si vous avez besoin de descendre très bas dans l'exploitation de Windows ou si vous générez des types de documents complexes avec des applications basées sur le modèle Document / Visualisation auquel cas il vaut mieux rester sous Borland C++.

1.2.3 C++ Builder contre VB

Depuis quelques années, Microsoft a proposé le langage Visual Basic associé à un environnement RAD pour écrire rapidement des applications Windows. En outre, il est intégré aux applications de la suite bureautique Office sous la forme VBA et permet de manipuler simplement les données de chaque application et offre une certaine souplesse pour effectuer des appels OLE entre ces applications.

S'il se révèle très agréable à utiliser pour concevoir un micro outil à usage unique ou une macro-commande dans l'un de vos logiciels préférés, son utilisation à vaste échelle devient, à mon avis, vite pénible. Deux syntaxes d'affectation, l'utilisation quasi systématique du fameux type variant, l'inefficacité des commandes arithmétiques sont autant d'arguments brandis par les détracteurs de Visual Basic (dont je suis) et qui, tout chauvinisme mis à part, écartent ce langage des développements importants.

Pour terminer cette partie, certains argueront (avec raison) de l'excellente interface entre Visual Basic et JET, le moteur de base de données de Microsoft sur lequel repose ACCESS. Il est vrai qu'ACCESS offre un environnement de programmation d'applications de bases de données particulièrement efficace. Toutefois, BDE, le moteur de base de données de Borland – dont l'API complète ainsi que des contrôles Windows orientés données est incluses dans C++ Builder – offre les mêmes fonctionnalités même si, contrairement à son concurrent, il est nécessaire d'invoquer un programme complémentaire (fourni) pour définir graphiquement une relation ou une requête.

1.3 Pour conclure

Souvenez vous que la programmation RAD est très axée sur la définition de l'interface et que, si elle permet de gagner un temps considérable sur le traitement des événements les plus simples, elle ne vous sera d'aucun intérêt voir même d'un certain désavantage lorsque vous voudrez composer des applications plus complexes. Toutefois, dans l'immense majorité des cas, je ne saurais trop recommander l'usage de tels outils.

2. L'environnement de développement C++ Builder

Après ces considérations somme toute assez philosophiques, nous allons attaquer désormais l'étude de C++ Builder à commencer par la prise en main de son interface identique à celle de son frère aîné : Delphi.

2.1 L'interface de C++ Builder

La figure 1 représente un exemple typique de l'interface de C++ Builder au cours d'une session de travail.

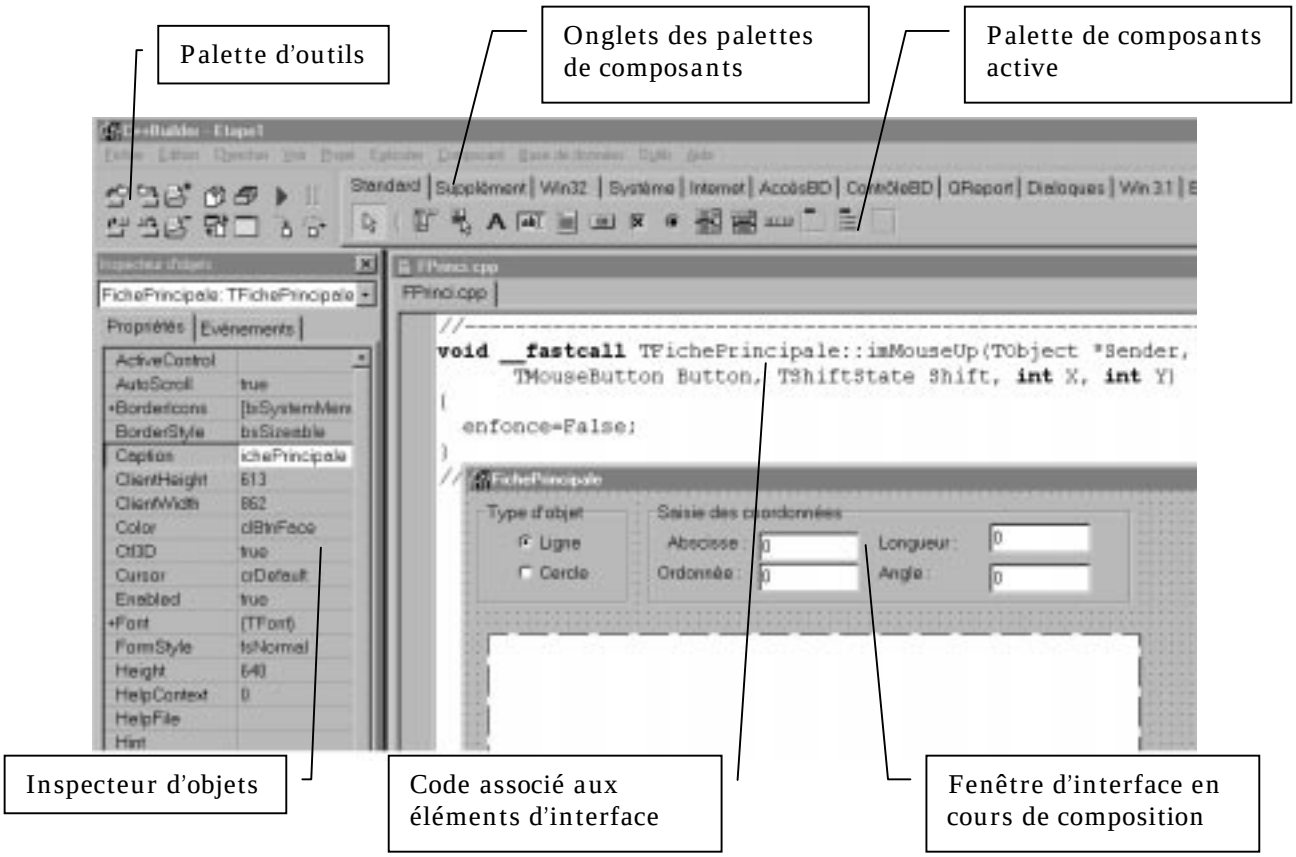


Figure 2.1 L'interface de C++ Builder

Cette interface est assez déroutante au premier abord car elle n'occupe pas tout l'écran. De fait, une partie des écrans des autres applications que vous utilisez (ou du bureau !) est visible.

On peut toutefois distinguer plusieurs grandes parties :

- La classique barre de menu
- La barre d'outils qui se décompose en 2 grandes parties :
  - La palette d'outils permettant d'effectuer les opérations les plus courantes (sauvegarde, ouverture de fenêtres, etc.)
  - Les palettes de composants disposées accessibles par des onglets

- L'inspecteur d'objets qui permet de manipuler les propriétés des composants et d'associer du code à leurs événements

- Les fenêtres d'interface créées par l'utilisateur. Ce sont les fenêtres de l'application en cours de création, elles portent ici le nom de *fiche* (form en anglais). Certaines peuvent être cachées, d'autres présentes à l'écran. On verra que dans la plupart des cas, leur position (ainsi que leurs autres caractéristiques géométriques) à l'exécution sont le reflet exact de ce qu'elle étaient lors de la conception

- L'éditeur de code. A chaque fiche ouverte correspond deux fichiers source (un fichier `.h` et un fichier `.cpp`) qui sont éditables dans cette fenêtre avec mise en évidence de la syntaxe.

D'autres fenêtres auraient pu être présentes dans des phases spécifiques de développement. Citons pêle-mêle : les inspecteurs de variables du débogueur, la liste des points d'arrêt, les différents experts, etc.

2.2 Les composantes de C++ Builder

Par défaut, C++ Builder utilise un compilateur C++, un éditeur de liens, un compilateur de ressources et un gestionnaire de projets intégrés. Il est toutefois possible de spécifier que vous désirez lui faire utiliser les outils en ligne de commande livrés conjointement ou même d'autres outils. Ce dernier cas, très intéressant lorsque l'on souhaite utiliser des modules compilés dans d'autres langages (c'est tout de même un peu technique) doit être étudié très soigneusement. En particulier, il faut s'assurer que les différents codes soient compatibles.

Théoriquement, C++ Builder se charge de gérer les imports et les exports des différentes librairies dynamiques (`DLL`) utilisées. Toutefois, vous pouvez gérer cela manuellement (notamment pour éviter qu'une de vos propres DLL n'exporte toutes ses définitions, ce qui est le comportement par défaut) en éditant manuellement le fichier des définitions (`.DEF`) puis en appelant l'utilitaire **`implib`**.

2.3 Création d'une application simple C++ Builder

C++ Builder permet de créer différents types de module très simplement en se laissant guider par des experts. Toutefois, il est possible de demander à créer une application simple en activant l'option **`Nouvelle application`** du menu **`Fichier`**. Les éléments automatiquement créés sont les suivants : une fiche nommée **`Form1`** ainsi que les fichiers associés **`Unit1.cpp`** et **`Unit1.h`**. Notons au passage que la terminologie unit est directement calquée sur celle chère à Delphi et que les fonctionnalités ainsi créées sont toujours renommables après coup.

Je recommande de toujours sauvegarder le projet juste après sa création : on évite ainsi la création des fichiers de compilation dans les répertoires par défaut. Cette opération est réalisée avec la commande **`Sauvegarder le projet sous…`** du menu **`Fichier`**. Le projet en lui même (fichier **`.bpr`**) est sauvegardé après les différents fichiers **`.cpp`** et **`.h`**.

Une fois cette opération réalisée, il reste à réaliser l'application en créant les objets d'interface et en leur associant des gestionnaires d'événements.

2.4 L'inspecteur d'objets et les propriétés

L'inspecteur d'objets est une fenêtre à deux volets respectivement spécialisés dans l'édition des valeurs des propriétés des composants et l'intendance de leurs gestionnaires d'événements.

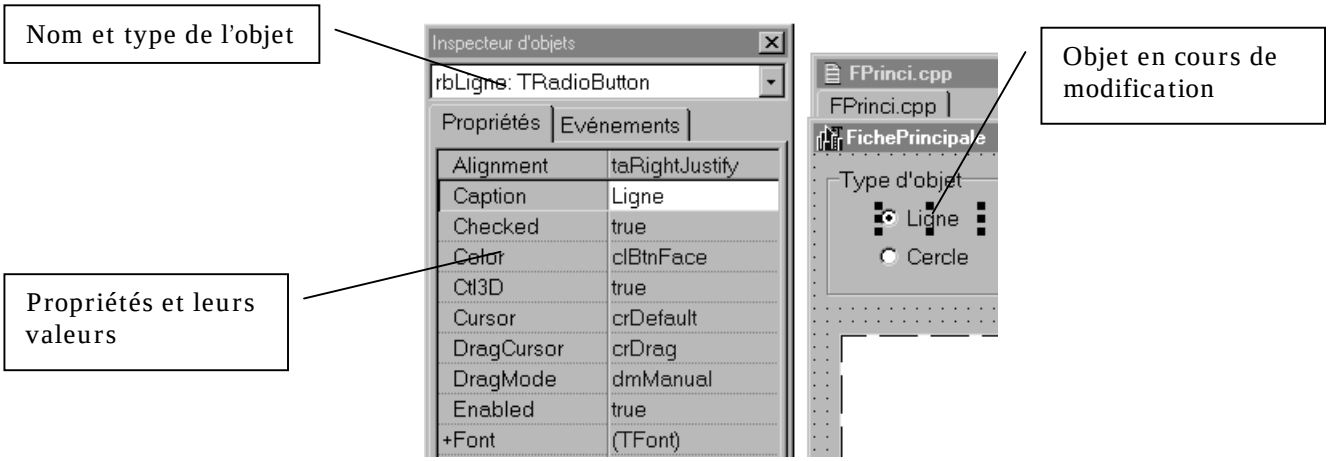


Figure 2.2 Edition des propriétés dans l'inspecteur d'objets

La figure précédente montre l'aspect de l'inspecteur d'objets lors de la modification des propriétés d'un bouton radio.

Selon le type de la propriété, l'édition se fera sous forme d'une boîte d'édition de texte simple, dans une liste à dérouler ou même une fenêtre spécialisée.

2.5 La propriété `Name`

Permettez-moi, s'il vous plait, de jouer au prof gavant (je sais, je fais ça très bien) et d'insister sur la propriété **`Name`**. Celle-ci est terriblement important car elle vous permet d'accéder à vos composants à l'intérieur de votre programme. Par défaut, lorsque vous ajoutez un composant à une fiche, C++ Builder lui confère un nom automatique du genre **`TypeNuméro`**, par exemple le premier label que vous poserez sur une fiche aura pour nom **`Label1`** ce qui n'est guère explicite !

Il est préférable de respecter certaines conventions de nommage de ses composants. Par exemple, il faut que le nom d'un composant vous indique :

- Le type du composant
- Sa fonction

L'usage veut que la fonction soit dans le corps du nom alors que le type est indiqué par un préfixe. La table suivante donne (à titre purement indicatif) quelques uns des préfixes utilisés dans ce manuel.

b :	bouton d'action	**lb** :	label
br :	bouton radio	**gb** :	boîte de groupe
cc :	case à cocher	**edit** :	boîte d'édition
pl :	panel	**me** :	mémo

Tableau 2.1 Quelques préfixes de nommage pour les composants VCL

2.6 Manipuler les événements

La manipulation des événements est quelque peu plus complexe. La figure suivante illustre quelques principes.

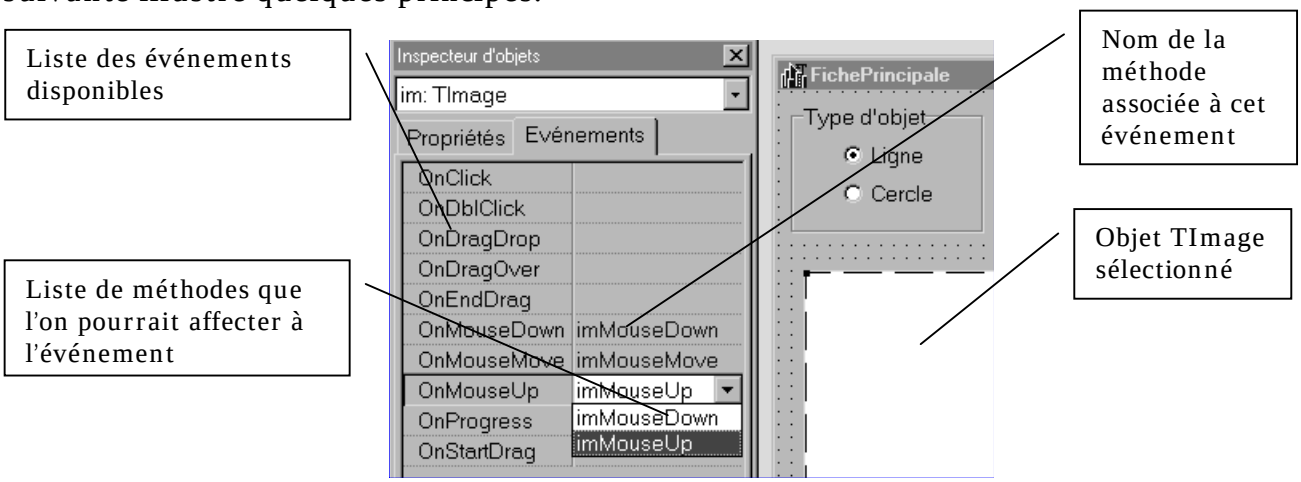


Figure 2.3 Manipulation des gestionnaires d'événements

Tout d'abord, il est important de rappeler certains principes :

Les gestionnaires d'événements sont toujours des méthodes de la fiche. En effet, avec C++ Builder, les événements générés par les contrôles sont toujours renvoyés vers la fiche.

Ce mécanisme (connu sous le nom de notification au parent) est très pratique car il permet de simplifier considérablement la tâche du programmeur : tous les événements pour une même fiche sont générés au même niveau. Néanmoins, il est parfois gênant. Prenons, par exemple, le cas de la gestion d'un groupe de boutons radio.

Typiquement, ces derniers seront rassemblés dans une fenêtre de groupe. Avec Borland C++, il était possible de rediriger les événements générés par les boutons radio vers la fenêtre de groupe et de les traiter ainsi avec une seule méthode. Avec la méthode C++ Builder, tous les événements étant automatiquement redescendus au niveau de la fiche, il faudra gérer individuellement les événements ou ruser comme un sioux.

Une même méthode peut gérer plusieurs événements si son prototype le permet.

Notons que la liste de paramètres d'un gestionnaire d'événements contient toujours au moins un paramètre nommé Sender, de type **`TObject*`** et qui contient l'adresse du composant ayant généré le message. D'autres paramètres peuvent être présents, par exemple :

- Les positions de la souris
- L'état des touches de modification du clavier

Pour créer une nouvelle méthode de gestion d'événement, il suffit de double cliquer dans l'espace vide à droite du nom de l'événement, une méthode avec le nom par défaut est alors créée. Son appellation reprend en partie le nom ou le type du contrôle générant l'événement et la dénomination de l'événement. Vous pouvez également choisir le nom vous même.

Pour affecter une méthode déjà existante à un gestionnaire, il suffit de puiser dans la liste déroulante.

2.7 C++ Builder et les exceptions

Le comportement de C++ Builder vis à vis des exceptions peut paraître parfois déroutant. En effet, la mise en place d'un gestionnaire d'exceptions, par exemple, pour l'exception **`EConvertError`** se traduit d'abord par l'affichage d'un message du genre :

Figure 2.4 Fenêtre d'interception d'une exception

Ce qui se traduit en clair par la phrase suivante : Le débogueur intégré à C++ Builder a intercepté l'interruption avant de vous passer la main. Notez au passage que la fenêtre contient le message d'explication inclus dans toute exception C++ Builder : il s'agit du texte entre guillemets Anglais.

Le plus important est de savoir que ce comportement n'existe que dans l'EDI. En effet, en exécution indépendante, ce message n'apparaîtrait que si l'exception déclenchée n'était pas traitée ; si vous fournissez un handler d'exception, celui-ci serait activé normalement.

Dans de nombreux cas, ce fonctionnement de l'EDI est plus une gène qu'un atout. Il est toutefois possible de le désactiver grâce à la boîte de dialogue – présentée sur la page suivante – directement issue du menu Outils → Options d'environnement, onglet Débogueur.

En plus de fournir de nombreuses options de configuration du débogueur, cette page nous permet de spécifier le comportement des exceptions lorsqu'un programme C++ Builder est lancé depuis l'EDI.

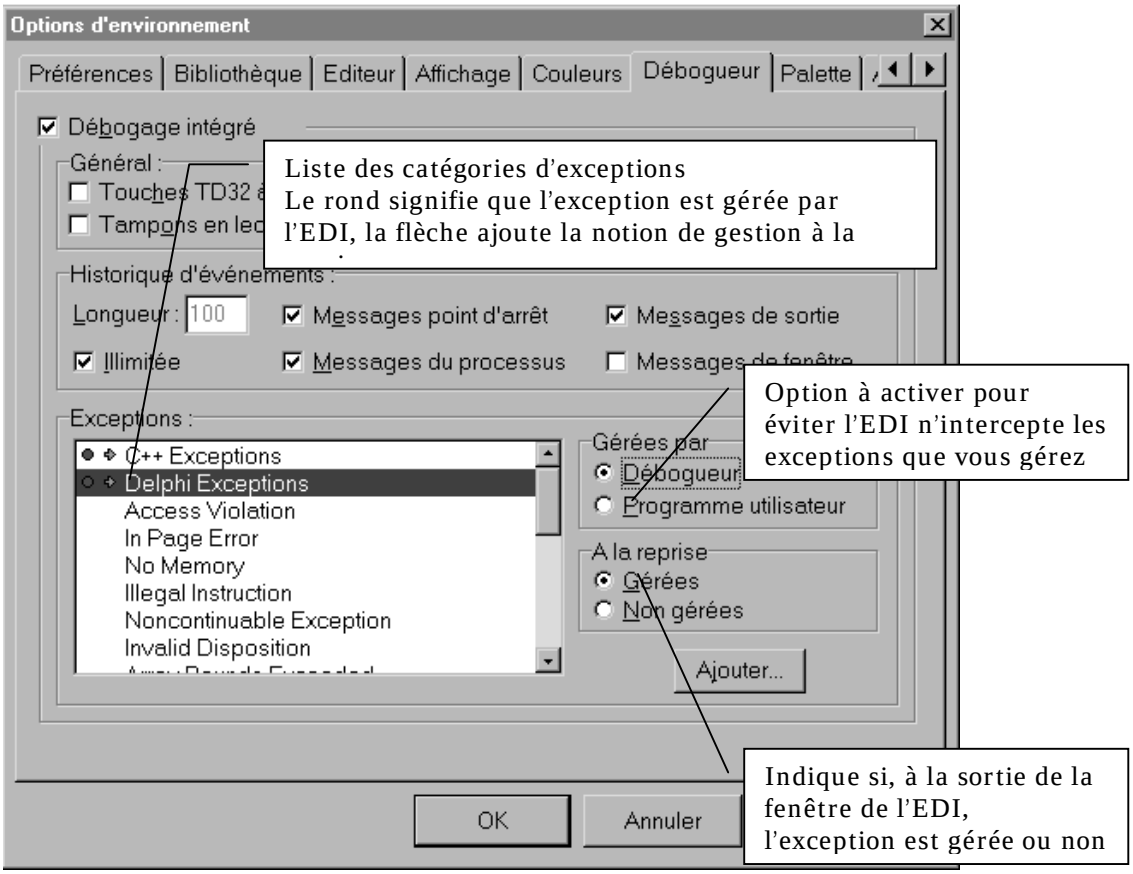


Figure 2.5 Options du débogueur de l'EDI C++ Builder

Ainsi, toutes les catégories marquées d'un petit rond (de couleur rouge !) – voir la figure précédente – sont gérées, d'abord par l'EDI et dans un second temps par le programmeur. Il est très facile de modifier ce comportement en sélectionnant le bouton radio Programme utilisateur.

En outre, si vous laissez l'EDI gérer votre interruption, il est possible de spécifier le comportement de votre programme à la sortie de la fenêtre de l'EDI en jouant sur les boutons radio « A la reprise ».

Par défaut, les exceptions de l'utilisateur sont rangées dans la rubrique « Exceptions C++ » alors que les exceptions levées par la VCL sont dans la catégorie « Exceptions Delphi » **et** dans la catégorie « Exceptions C++ ».

Le seul moyen d'être tranquille consiste donc à désactiver l'interception par l'EDI de tout type d'exception.

2.8 Utilisez la fenêtre d'historique !

La fenêtre d'historique des événements est l'un des mécanismes de traçage des programmes les plus méconnus de Windows. C++ Builder nous permet de consulter l'état de cette fenêtre en l'activant dans le menu Voir.

La page de configuration des options du débogueur permet également de modifier le comportement de cette fenêtre. Focalisons nous donc sur ce dernier aspect :

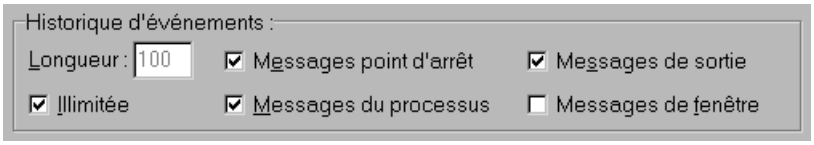


Figure 2.6 Options de la fenêtre d'historique

Deux grands types d'options sont gérables :

- La taille de l'historique
  - Illimité
  - Nombre de lignes spécifiable par l'utilisateur
- Les catégories de message tracés
  - **Les points d'arrêt** : à chaque fois que l'exécution est stoppée à un point d'arrêt, un message est imprimé dans la fenêtre
  - **Les messages du processus** : ils concernent, par exemple, le chargement de DLL ou d'informations de déboguage au lancement du programme
  - **Les messages de fenêtre** : à chaque fois qu'une fenêtre reçoit un message, il y a émission d'un message contenant :
    - Le récepteur du message
    - Les paramètres (WPARAM et LPARAM)

Attention, si ce comportement peut se révéler très pratique, il peut créer des traces énormes. En effet, on imagine rarement le volume considérable de messages traités par les applications Windows.

- **Les messages de sortie** sont créés par l'utilisateur à l'aide de la commande API **`OutputDebugString`**. Le seul problème tient à la bufferisation de cette fenêtre. En effet, les messages envoyés par **`OutputDebugString`** ne sont pas affichés immédiatement, ce qui limite l'intérêt de cette fonctionnalité.
- En outre, cette fonction prend comme paramètre un **`char *`** et non pas une **`AnsiString`** comme la plupart des méthodes et fonctions de la VCL, ce qui rend des **`cast`** nécessaires.

3. Etude de la VCL

3.1 Organisation de la VCL

La VCL (Visual Component Library) livrée par Inprise avec C++ Builder ou Delphi est un ensemble de classes orientées vers le développement rapide d'application. Toutes les classes présentes partagent un ancêtre commun : la classe **`TObject`**. Elles possèdent également une caractéristique particulière : elles ne peuvent pas posséder d'instances statiques : seules les instances dynamiques créées avec **`new`** sont acceptées. Ceci est nécessaire pour assurer la compatibilité avec Delphi qui ne reconnaît que les instances dynamiques. Toutes les classes de la VCL sont implémentées en Pascal Objet. En fait, la librairie d'exécution de C++ Builder est celle de Delphi, ce qui implique un certain nombre de gymnastiques pour assurer une édition de liens correcte.

3.2 Les composants

Les composants sont des instances de classes dérivant plus ou moins directement de **`TComponent`**. Si leur forme la plus habituelle est celle des composants que l'on dépose d'une palette vers une fiche, ils englobent plus généralement la notion de brique logicielle réutilisable. Bien que non déclarée virtuelle pure[1] la classe **`TComponent`** n'est pas destinée à être instanciée. Compulsons sa documentation ;nous y apprenons que la plupart des méthodes sont protégés, c'est à dire inaccessibles à l'utilisateur, c'est une technique courante en Pascal Orienté Objet : définir le cadre de travail à l'aide de méthodes virtuelles protégées, lesquelles seront déclarées publiques dans les classes dérivées.

Autre aspect particulièrement intéressant : la présence des méthodes **`AddRef`**, **`Release`** et **`QueryInterface`** ce qui dénote l'implémentation de l'interface OLE **`IUnknown`**. Ainsi, lorsque l'on transformera un composant VCL en composant ActiveX, la gestion d'**`IUnknown`** sera directement prise en compte au niveau de **`TComponent`**. De la même manière, **`TComponent`** implémente l'interface principale d'automation **`IDispatch`** (méthodes **`GetIDsOfNames`**, **`GetTypeInfo`**, **`GetTypeInfoCount`** et **`Invoke`**). Nous aurons l'occasion de revenir plus en détails sur ces mécanismes spécifiques à OLE dans un prochain chapitre.

Pour finir, notons que les composants que vous créez avec Delphi ou C++ Builder sont compatibles avec les deux environnements : autrement dit, il est tout à fait possible d'utiliser dans C++ Builder un composant créé avec Delphi et réciproquement.

---

[1] En effet, la classe **`TComponent`**, à l'instar de toutes les autres classes de la VCL est implémentée en langage Pascal Objet, lequel s'accommode assez mal de la notion de classe virtuelle pure.

3.3 Les Contrôles

On appellera Contrôle, tout objet instance d'une classe dérivant de **`TControl`**. Les contrôles ont pour caractéristique particulière d'être des composants à même de s'afficher sur une fiche dans leurs dimensions d'exécution.

Par exemple, les boutons (**`TButton`**), les étiquettes (**`TLabel`**) ou les images sont des contrôles. En revanche, les menus (**`TMenu`**) ou les boîtes de dialogue communes (**`TCommonDialog`**) de Windows qui sont représentées par une icône sur les fiches ne sont pas des contrôles.

En terminologie C++ Builder, les contrôles se séparent en deux grandes catégories :

- Les contrôles fenêtrés
- Les contrôles graphiques

3.3.1 Les contrôles fenêtrés

Comme leur nom l'indique, les contrôles fenêtrés sont basés sur une fenêtre Windows. Ceci leur confère plusieurs caractéristiques :

- Ils disposent d'un **`handle`** de fenêtre. Un **`handle`** est un numéro unique alloué par le système à toute ressource telles que les fenêtres, les polices, les pinceaux ou les brosses.

  De ce fait, chaque contrôle fenêtré monopolise une ressource fenêtre : il ne pourra y en avoir qu'un nombre limité présents dans le système à un instant donné.

- Leur état visuel est sauvegardé par le système. Lorsqu'ils redeviennent visibles, leur apparence est restaurée automatiquement sans que le programmeur n'ait à s'en soucier.

  Conséquence négative : il est plus lent de dessiner dans un contrôle fenêtré que dans un contrôle graphique.

- Ils peuvent recevoir la focalisation, c'est à dire intercepter des événements en provenance du clavier

- Ils peuvent contenir d'autres contrôles. Par exemple, les boîtes de groupe (**`TGroupBox`**) sont des contrôles fenêtrés.

- Ils dérivent de la classe **`TWinControl`** et, la plupart du temps, de la classe **`TCustomControl`** laquelle dispose d'une propriété **`Canvas`** permettant de dessiner facilement dans la zone client du contrôle. Si vous devez dessiner dans la zone client d'un contrôle sans que celui-ci nécessite la focalisation, alors, il faudra mieux utiliser un contrôle graphique moins gourmand en ressources système.

La plupart des éléments actifs dans une interface sont des contrôles fenêtrés. En particulier, les fiches sont des contrôles fenêtrés.

3.3.2 Les contrôles graphiques

Contrairement aux contrôles fenêtrés, les contrôles graphiques ne sont pas basés sur une fenêtre. Ils ne disposent donc pas d'un **`handle`** de fenêtre et ne peuvent recevoir d'événements en provenance du clavier (pas de *focalisation*). Ils ne peuvent pas non plus contenir d'autres contrôles, toutes ces fonctionnalités étant réservées aux contrôles fenêtrés (voir paragraphe précédent). Du coup, ils sont moins gourmands en ressources que les composants fenêtrés.

En outre, l'état visuel (ou apparence graphique) d'un contrôle graphique n'est pas gérée par le système :

Il sera plus rapide de dessiner dans un contrôle graphique que dans un contrôle fenêtré car dans ce cas, seul l'affichage est concerné. Il n'y a pas de sauvegarde de l'état dans la mémoire du système.

Lorsqu'un contrôle graphique est masqué puis réaffiché, le système lui envoie un événement **`WM_PAINT`** lui indiquant qu'il doit mettre à jour son apparence visuelle. Le programmeur doit donc intercepter cet événement (**`OnPaint`**) pour redessiner la zone client de son composant. Pour cela, il dispose de la propriété **`Canvas`** qui fournit une plate-forme de dessin des plus agréables à utiliser. Il faut également savoir que le gestionnaire d'événement **`OnPaint`** est appelé par la méthode virtuelle **`Paint`** directement déclenchée par l'événement Windows **`WM_PAINT`**. De fait, la création d'un nouveau contrôle graphique passe le plus souvent par la redéfinition de cette méthode.

La plupart des composants purement orientés vers l'affichage sont des contrôles graphiques. Citons par exemple **`TLabel`** (affichage pur et simple d'un texte), **`TImage`** (affichage d'un graphique) ou **`TBevel`** (affichage d'une ligne, d'une forme en creux ou en relief).

3.4 Les boîtes de dialogue standard de Windows

Les boîtes de dialogue standard de Windows sont des objets partagés par toutes les applications et permettant d'effectuer des opérations de routine telles que la sélection d'un nom de fichier, la configuration de l'imprimante ou le choix d'une couleur. C++ Builder encapsule la plupart d'entre elles dans des classes non visuelles dont les composants sont regroupés au sein de la palette Dialogues. Les classes associées dérivent de **`TCommonDialog`**, classe servant à établir le lien entre les classes de C++ Builder et les ressources Windows incluses dans la DLL `COMMONDLG.DLL`.

Ces composant ne sont pas des contrôles : ils n'apparaissent pas sous leur forme définitive lorsqu'ils sont insérés sur une fiche mais plutôt au travers d'une icône. Cette dernière n'apparaît pas lors de l'exécution, au contraire de la boîte qui elle sera invoquée à l'aide de sa méthode **`Execute`**. L'icône posée sur la fiche ne sert qu'à réserver une ressource Windows et permet de modifier les propriétés associées à la boîte. La figure suivante illustre la palette dialogues.

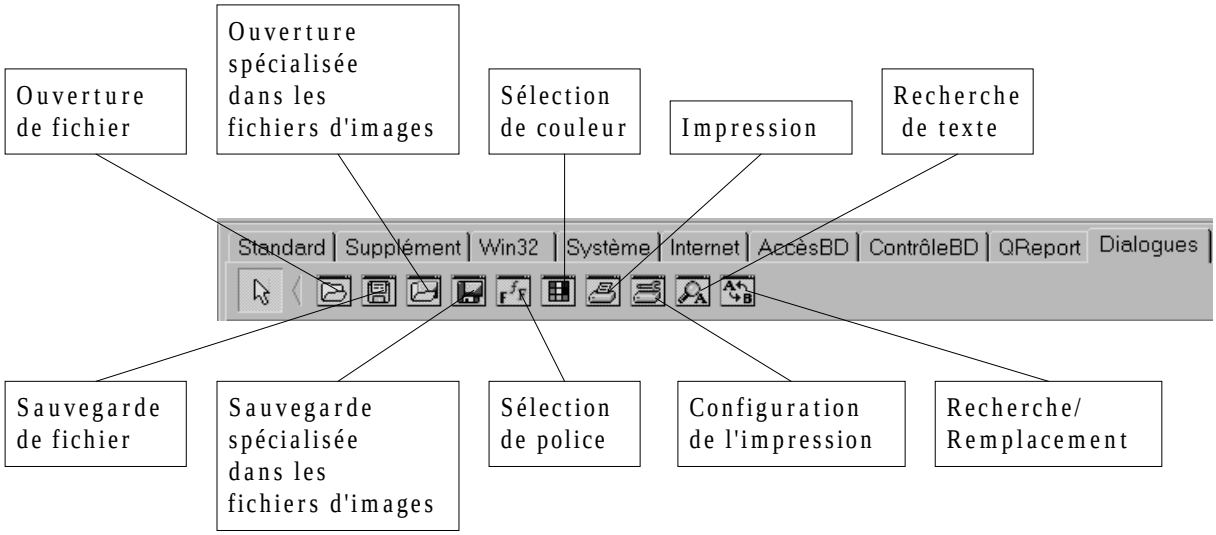


Figure 3.1 La palette dialogues

Ces boîtes de dialogue sont toujours exécutées en mode *modal* à l'aide de leur méthode **`Execute`**. Cela signifie qu'elles monopolisent la souris et le clavier tant que l'utilisateur ne les a pas fermées. Le résultat de **`Execute`** indique à l'invocateur à l'aide de quel bouton l'utilisateur à fermé la boîte. Les constantes (aux noms très explicites) sont les suivantes :

`mrOk`	**`mrCancel`**	**`mrNo`**	**`mrAbort`**
`mrRetry`	**`mrIgnore`**	**`mrYes`**	

Tableau 3.1 Codes de retour des boutons modaux

Insistons sur le fait que l'utilisation de ces classes dépend fortement de la DLL standard de Windows `COMMDLG.DLL`, leur aspect et leur comportement peuvent donc varier légèrement d'un système à un autre.

Nous ne détaillerons ici que l'utilisation des boîtes de manipulation de fichiers et de sélection de couleur.

3.4.1 Les boîtes de dialogue de manipulation de fichiers

Il est particulièrement pratique d'utiliser les boîtes de dialogue communes de Windows pour récupérer le nom d'un fichier à ouvrir ou à enregistrer car elles permettent, en particulier, une gestion intégrée de la navigation dans l'arborescence des répertoires. Les classes **`TOpenDialog`** et **`TSaveDialog`** sont très similaires dans le sens où elles reposent sur les mêmes propriétés. Elles ont d'ailleurs une structure de donnée identique. Voici un exemple de boîte d'ouverture :

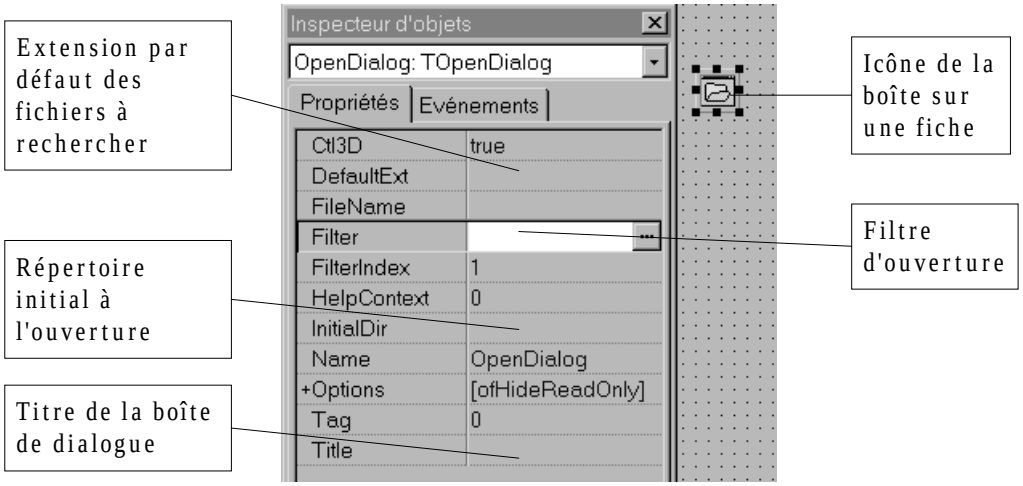


Figure 3.2 Propriétés des dialogues orientés fichier

Quelques propriétés méritent plus ample discussion :

- **`FileName`** contient, avant exécution, le nom du fichier à ouvrir par défaut, et au retour de l'exécution, le nom résultat de la recherche

- **`Filter`** contient la liste des formats de fichiers (spécifiés par leurs extensions) que l'application est susceptible d'ouvrir. En l'absence de toute indication, « *.* » est assumé. Cette propriété a un format bien particulier. C'est une chaîne de caractères obéissant au format :

  Chaîne de description|*liste des extensions associées*|*description* ...

Bien que l'on puisse très bien la saisir ainsi, il vaut mieux se reposer sur la boîte spéciale qui apparaît lorsque l'on clique sur ...

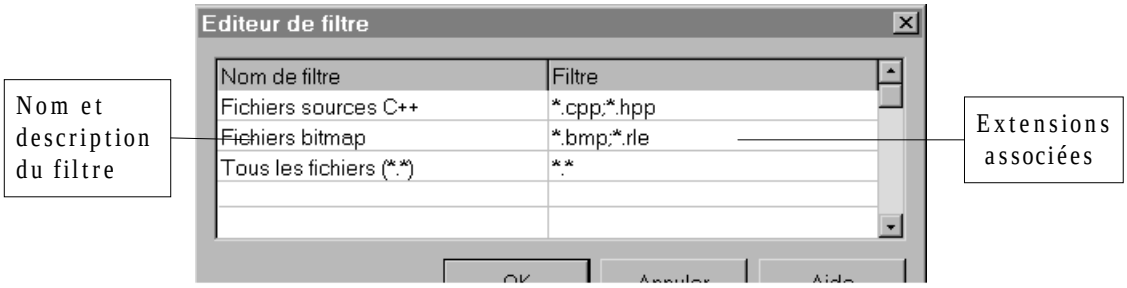


Figure 3.3 Boîte d'édition de la propriété filter

Les deux boîtes (ouverture et fermeture) se distinguent essentiellement par les options qui sont disponibles (précisément dans la propriété **`Options`**). Par exemple, la boîte d'ouverture (**`TOpenDialog`**) peut présenter une case à cocher proposant d'ouvrir un fichier en lecture seulement.

Il existe deux versions spécialisées dans l'ouverture et la sauvegarde de fichiers image qui ne posent pas de problème spécifique.

3.4.2 La boîte de sélection de couleurs

La figure suivante montre les propriétés les plus intéressantes de la boîte de sélection de couleurs de C++ Builder (classe **`TColorDialog`**).

Les options sont particulièrement intéressantes car elles conditionnent toute l'exécution, et en particulier la possibilité d'offrir à l'utilisateur la potentialité de créer lui-même ses propres couleurs.

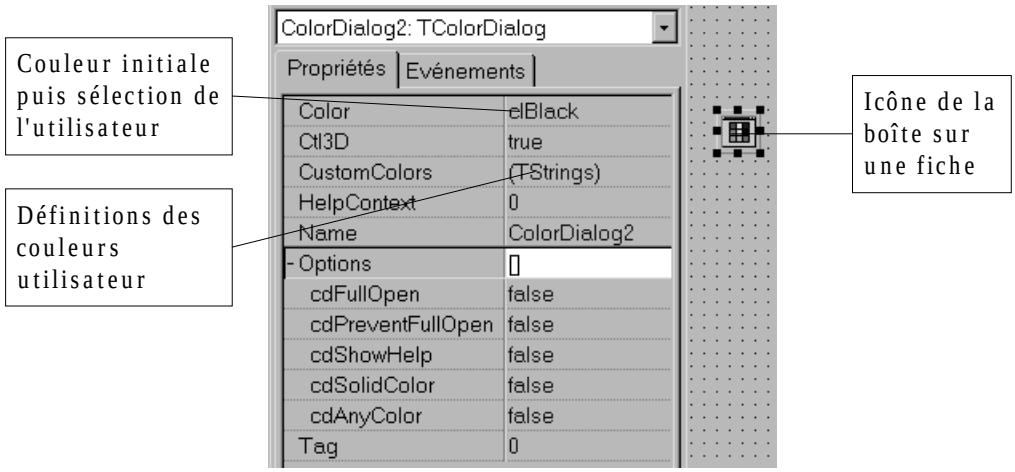
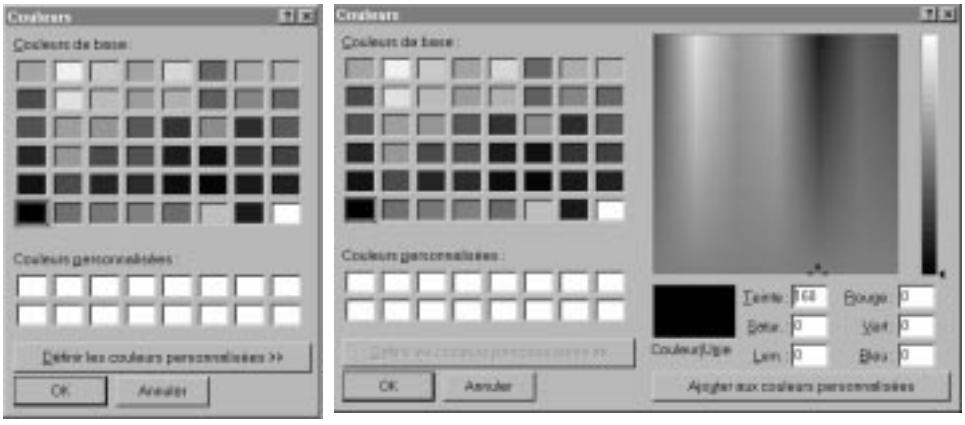


Figure 3.4 Propriétés de la boîte de choix de couleur

Une fois n'est pas coutume, intéressons nous aux options internes de la boîte :

`cdFullOpen` : dès l'ouverture, la boîte est en mode complet, c'est à dire avec la possibilité de définir les couleurs utilisateurs et de choisir parmi l'intégralité du nuancier. Sinon, la boîte s'ouvre en mode standard. La figure suivante montre les deux modes de fonctionnement de la boîte.



(a) : Mode standard (a) : Mode complet

Figure 3.5 Les deux modes de fonctionnement de la boîte de sélection de couleurs

cdPreventFullOpen : si cette option est activée, alors il est impossible d'activer le mode complet. Le bouton « Définir les couleurs personnalisées » du mode standard est caché

cdShowHelp : si cette option est activée, alors un bouton permettant d'activer un texte d'aide est présent

cdSolidColor et **cdAnyColor** sont deux options qu'il est utile d'activer si la carte graphique de l'utilisateur final ne supporte pas le mode *true colors*. Lorsque **cdSolidColor** est à **true**, alors la sélection d'une couleur obtenue par tramage renvoie vers la vraie couleur la plus proche. Si **cdAnyColor** est activée, alors il est possible de créer de nouvelles couleurs par tramage.

3.4.3 La boîte de sélection de Fonte

Cette boîte, très simple, permet de sélectionner une police parmi celles installées dans Windows. Toutes les informations sélectionnées sont regroupées dans la propriété Font qui elle même peut se déplier pour donner des informations aussi diverses que : le nom de la police, le jeu de caractères employé, le crénage, le style, la couleur ou, tout simplement, la hauteur des caractères.

La figure suivante illustre l'ensemble des propriétés disponibles :

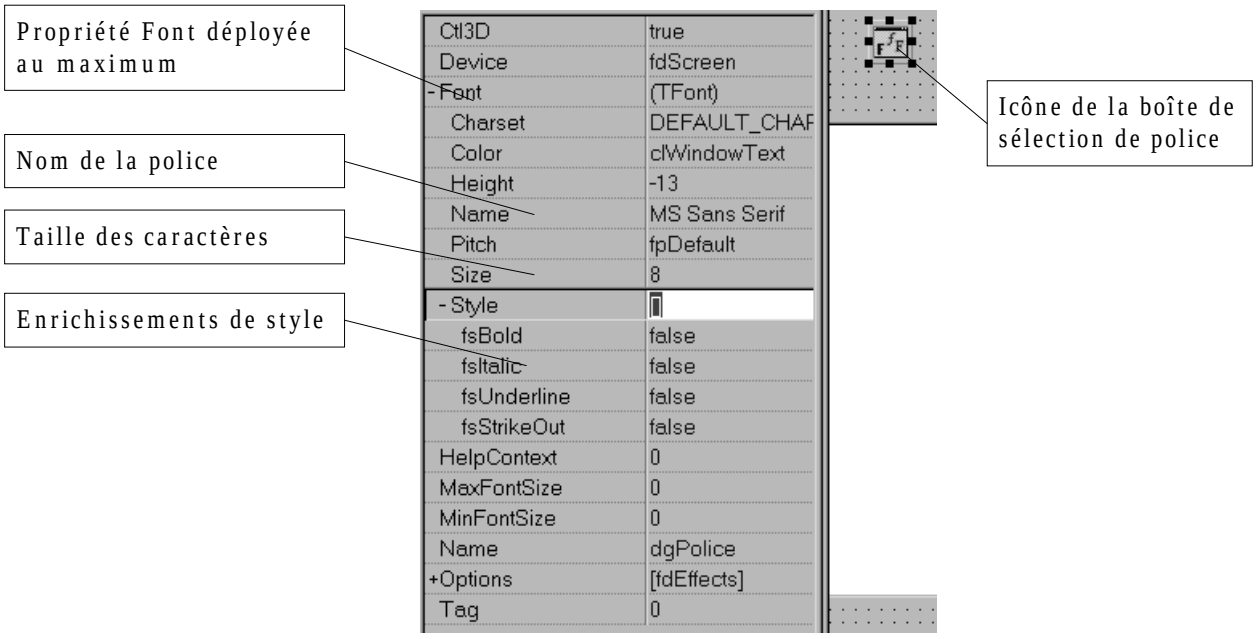


Figure 3.6 Propriétés de la boîte de sélection de police

Les options de la boîte elle même permettent de limiter l'utilisateur dans ces choix (par exemple, ne l'autoriser qu'à choisir des fontes non proportionnelles) ou de contrôler l'aspect visuel de la boîte.

3.4.4 Les boîtes de Recherche et Recherche / Remplacement

Ce sont les boîtes les plus compliquées à utiliser. Tout d'abord, ce sont les seules pour lesquelles le comportement n'est pas modal et binaire :

En effet, elles restent ouvertes tant que l'utilisateur appuie sur le bouton « Suivant ». En outre elles n'ont pas de bouton « Ok », le seul moyen permettant de les fermer consiste à utiliser le bouton « Annuler ».

En outre, elles n'effectuent pas directement la recherche ! Elles ne permettent que de saisir le texte à rechercher (ou le texte de remplacement) et de sélectionner des options. Le corps de la recherche reste à la charge du programmeur.

Les options de ces boîtes de dialogue sont aisées à comprendre, elles conditionnent l'affichage des diverses possibilités et le résultat de la sélection de l'utilisateur.

Le résultat de la boîte comprend, entre autres : la recherche de mots complets (**`frWholeWord`**), la prise en compte de la casse (**`frMatchCase`**), le sens et la portée de la recherche (**`frDown`**, **`frHideUpDown`**, etc.) .

Par défaut, les boîtes s'affichent avec l'étendue totale de leurs possibilités. Toutefois, l'utilisateur peut les limiter en activant des options dont le nom comprend **`Hide`** ou **`Disable`**, par exemple **`frDisableMatchCase`** désactive la case permettant relative à la casse des mots recherchés.

Comme la recherche elle même, la prise en compte des options de recherche est entièrement à la charge du programmeur. On ne répètera jamais assez que ces boîtes ne sont que des outils de saisie des options !

La boîte de recherche simple fournit un événement nommé **`OnFind`** notifiant au programmeur l'appui sur la touche « Suivant » de la boîte. La boîte de remplacement lui ajoute un événement nommé **`OnReplace`**.

Un exemple d'utilisation consiste à utiliser ces événements en conjonction avec la méthode **`FindText`** d'un composant **`TRichEdit`**.

3.4.5 Exercice sur les boîtes de dialogue communes (✶)

Enoncé

Nous vous proposons de tester l'utilisation des différentes boîtes de dialogue sur le texte contenu dans un composant **`TRichEdit`**. Ce composant est très proche de **`TMemo`** dans le sens où il reprend la plupart de ses fonctionnalités en ajoutant celle de lire un fichier **`.rtf`** et de le formater correctement. Malheureusement, il ne propose pas ni propriété ni méthode permettant d'accéder à la mise en forme de chaque caractère. En revanche, il est possible de formater globalement le texte.

Les fonctionnalités suivantes devront être accessibles à partir d'un menu :

- Charger le texte du mémo (méthode **`LoadFromFile`** de la propriété **`Lines`**) à partir d'un fichier sélectionné à l'aide d'une boîte commune d'ouverture.

  Les extensions possibles en lecture sont :

  - .obc (Fichier de type OBC – en fait du texte !) par défaut

- .txt (Fichier de type texte)
- .cpp, .h et .hpp (Fichiers de type C++ Builder)

- Sauvegarder le texte du mémo (méthode **`SaveToFile`** de la propriété **`Lines`**) dans un fichier sélectionné à l'aide d'une boîte commune de sauvegarde. Seule l'extension .obc est disponible à la sauvegarde
- Manipuler l'apparence visuelle du texte (propriété **`Font`** du mémo)
  - Changement de couleur (boîte de sélection de couleurs)
  - Changement de police (boîte de sélection de fonte)
- Effectuer des recherches et des opérations de recherche / remplacement grâce aux boîtes idoines.
- Quitter l'application !

Rien dans cet exercice n'est compliqué ! il y a juste beaucoup de code à écrire …

Pour aller plus loin dans l'exercice (✶✶) :

Proposer de sauvegarder le texte s'il a été modifié par des opérations de recherche / remplacement ou par une saisie utilisateur avant d'en charger un nouveau ou de quitter l'application (l'événement **`OnCloseQuery`** de la fiche est à considérer).

Solution partielle

La figure suivante illustre l'aspect de l'interface à construire :

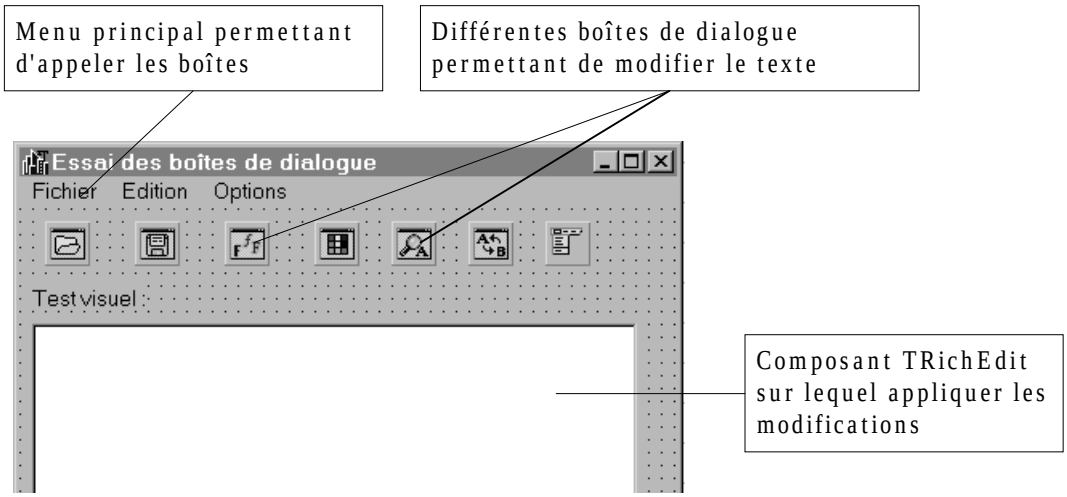


Figure 3.7 Interface de l'exercice sur les boîtes de dialogue

3.5 Les boîtes combo

Les boîtes combo permettent de rassembler en un seul composant les fonctionnalités d'une zone de saisie et d'une boîte de liste moyennant quelques limitations :

- La zone d'édition ne peut pas avoir de masque de saisie
- La boîte de liste est à sélection simple

Du coup, les propriétés sont hybrides de celles de la boîte de liste et de la zone d'édition, pour citer les plus importantes :

`Items` : liste des éléments présents dans la liste déroulante

`ItemIndex` : numéro de l'élément sélectionné. Le texte de l'élément sélectionné est accessible via : **`Items->Strings[ItemIndex]`** ou plus simplement via **`Text`** car le texte de l'élément sélectionné est recopié dans la zone d'édition

`Text` : texte sélectionné dans la liste ou saisi dans la zone d'édition

`Sorted` : à l'instar des boîtes de liste « normales », il est possible de stipuler que les différents éléments de la liste soient triés dans l'ordre lexicographique croissant.

`Style` : propriété déterminant le style de la boîte combo

- **`csSimple`** : la liste est visible en permanence
- **`csDropDown`** : il s'agit de la présentation habituelle d'une boîte combo où l'affichage de la liste s'obtient par activation d'un bouton placé à droite de la zone d'édition.
- **`csDropDownList`** : identique à la précédente à la différence que l'on ne peut sélectionner qu'un élément déjà présent dans la liste. Cela permet de créer une liste fixe que l'on fait dérouler à l'aide du bouton.
- **`csOwnerDrawFixed`** : l'affichage de chaque cellule de la liste est sous la responsabilité du programmeur qui doit fournir une réponse à l'événement **`OnDrawItem`** appelé pour dessiner chacun des éléments de la liste. Ceci permet, par exemple, de créer une liste de couleurs ou de styles de traits (voir exercice). Dans ce cas, les différents éléments de la liste sont tous de la même hauteur, laquelle est fixée par la propriété **`ItemHeight`**.
- **`csOwnerDrawVariable`** : identique à la précédente à la différence que chaque élément peut avoir une hauteur différente. L'événement **`OnMeasureItem`** est lancé juste avant **`OnDrawItem`** afin de fixer la hauteur de l'élément à dessiner.

La possibilité offerte au programmeur de contrôler totalement l'affichage de chacun des éléments de la liste est d'autant plus intéressant que cela n'empêche pas pour autant la saisie de nouveaux éléments dans la zone de saisie. Cela permet, par exemple d'afficher un petit en regard des derniers éléments saisis.

Il est important de noter que la saisie d'un nouvel élément n'entraîne pas pour autant son inclusion dans la liste des possibilités. Cette dernière est laissée à la

responsabilité du programmeur. Une fois de plus, cela s'avère particulièrement judicieux car l'on a alors la possibilité de « filtrer » les inclusions.

A la conception, il est possible de fixer le texte initial d'une boîte combo grâce à sa propriété **`Text`** mais pas en spécifiant un numéro d'index dans la liste. Ceci est seulement réalisable par programmation, par exemple dans l'événement **`OnCreate`** de la liste.

3.5.1 Exercice résolu n°1 (✶)

Enoncé :

On désire réaliser une boîte combo qui n'accepte d'inclure dans sa liste que des nombres compris entre 0 et une limite indiquée dans une zone d'édition. Réaliser cette interface sachant que la validation de la valeur se fait par l'appui sur un bouton.

Solution partielle :

Puisque la validation de la saisie se fait grâce à un bouton, nous allons nous concentrer sur le code de l'événement **`OnClick`** associé. La limite de saisie est située dans un attribut de la classe fiche nommé `limiteSaisie`.

```
void __fastcall TFSaisie::bnValideClick(TObject *Sender)
{
  // Tout d'abord, on essaye de transformer la valeur chaîne en nombre entier
  try
  {
    int valeur=cbSaisie->Text.ToInt();
    // Ouf ! c'est un entier ! S'il est dans la limite on l'ajoute a la liste
    if ((valeur > 0) && (valeur < valeurLimite))
    {
      cbSaisie->Items->Add(cbSaisie->Text);
      meTrace->Lines->Add("La valeur "+cbSaisie->Text+" est ajoutée");
    }
    else
      meTrace->Lines->Add("Valeur "+cbSaisie->Text+" en dehors des limites");
  }
  // Test de l'erreur de conversion
  // le texte saisi est effacé après envoi d'un message
  catch (EConvertError &e)
  {
    ShowMessage("Saisie Incorrecte : "+e.Message);
    cbSaisie->Text="";
  }
}
```

Programme 3.1 Ajout d'une ligne saisie dans une boîte combo

Commentaires :

- Notez l'utilisation de l'exception **`EConvertError`** qui est lancée à chaque conversion ratée. La propriété Message contient un message d'erreur utile à des fins de diagnostique.

- La classe **AnsiString** contient des *méthodes* **ToInt** et **ToDouble** que l'on peut utiliser en lieu et place des *fonctions* de conversion **StrToInt**, et **StrToFloat**.

Pour continuer l'exercice :

Utiliser l'événement **OnChange** d'une zone d'édition couplé à une gestion d'exception pour modifier la valeur limite d'inclusion.

La solution se trouve dans l'unité `Usaisie` du répertoire `Combos` du Power Bear Pack™

3.5.2 Exercice résolu n°2 (✶✶)

Enoncé

On désire créer une boîte combo permettant de sélectionner un style de trait pour un dessin. En outre, chaque trait est dessiné au milieu de son élément, lequel est de taille fixe. On vérifiera préalablement dans l'aide que 6 styles sont disponibles : ils correspondent à des constantes numériques s'échelonnant de 0 à 5 (suivre **TPen** → Propriété Style)

Solution partielle

Pour résoudre cet exercice il convient de se rendre compte que la boîte combo doit contenir autant d'éléments que l'on doit en tracer. Cela peut paraître une évidence mais il faut tout de même y penser. Aussi, lors de la phase de conception de l'interface, vous n'oublierez pas de poser 6 éléments dans la propriété **Items** de votre boîte combo (par exemple, les nombres de 0 à 5).

Ensuite, il suffit de répondre à l'événement **OnDrawItem** comme le montre le code suivant, non sans avoir au préalable fixé le style de la boîte combo à **csOwnerDrawFixed** !

```
void __fastcall TFComboTraits::cbStyleDrawItem(TWinControl *Control,
          int Index, TRect &Rect, TOwnerDrawState State)
{
  // Récupération d'un pointeur sur la boîte combo
  TComboBox *cb=dynamic_cast<TComboBox *>(Control);
  // On convertit en TPenStyle l'index passé en argument
  TPenStyle  leStyle=static_cast<TPenStyle>(Index);
  // Récupération d'un pointeur sur le canevas de dessin de la combo
  TCanvas   *cv=cb->Canvas;
  // Hauteur du tracé
  int        vert=(Rect.Top+Rect.Bottom) >> 1;

  // Remplissage de toute la zone de tracé avec un carré blanc
  cv->Brush->Style=bsSolid;
  cv->Brush->Color=clWhite;
  cv->FillRect(Rect);

  // Affichage du motif de dessin
  cv->Pen->Color=clBlack;
  cv->Pen->Style=leStyle;
  cv->MoveTo(Rect.Left,vert);
```

```
  cv->LineTo(Rect.Right,vert);
}
```

Programme 3.2 Affichage de styles de traits dans une boîte combo

Commentaires :

Etudions les paramètres du gestionnaire **`OnDrawItem`** :

- **`TWinControl Control`** : boîte combo ayant émis le message. Comme nous voulons travailler le plus générique possible, nous allons transformer ce paramètre en pointeur sur une boîte combo. Le code le plus simple serait :

  ```
  TComboBox *cb=(TComboBox *)Control
  ```

  Hors, en bon programmeur objet, vous savez très bien qu'il est très dangereux d'effectuer des conversions de promotion (aussi appelées *downcast*). Utiliser l'opérateur C++ **`dynamic_cast`** permet de sécuriser l'opération.

- **`int Index`** : numéro de l'élément à dessiner. Dans notre cas, nous nous dépêchons de le convertir en **`TPenStyle`** (l'utilisation de **`static_cast`** vous montre un second opérateur de conversion du C++) afin de l'utiliser pour positionner l'option **`Style`** de l'outil **`Pen`** du canevas de la boîte combo.

- **`TRect &Rect`** : zone rectangulaire où doit s'effectuer l'affichage. Celui-ci est supporté par le canevas (contexte d'affichage) de la boîte combo, lequel est fourni par la propriété **`Canvas`** de **`TComboBox`**. La zone rectangulaire est caractérisé par les quatre coordonnées **`Top`**, **`Bottom`**, **`Left`** et **`Right`** qui se passent de commentaire.

- Le dernier paramètre indique si l'on est vraiment sur qu'il faille redessiner. Et contrairement au proverbe, « dans le doute : redessine ! »

Vous noterez que l'on a pris soin d'effacer la zone de travail en la recouvrant d'un rectangle blanc (grâce à la méthode **`FillRect`**). Ceci permet de s'assurer que l'on démarre bien sur une surface vierge. En effet, lorsque vous faites défiler la liste, les différents éléments doivent s'afficher au même endroit ce qui occasionnerait des collisions néfastes

Solution complète :

La solution se trouve dans l'unité `UComboTraits` du répertoire `Combos` du Power Bear Pack™

3.5.3 Exercice résolu n°3 (✶✶)

Enoncé

On désire créer une boîte combo permettant de sélectionner l'épaisseur du trait pour un dessin. Il est également stipulé que la hauteur de chaque élément doit être de

5 fois l'épaisseur du trait qui est dessiné au milieu de son élément. Les épaisseurs étudiées devront évoluer de 0 à 10 compris. Que peut-on dire de l'épaisseur 0 ?

Solution partielle :

Cet exercice se résout facilement dès lors que l'on a compris le précédent. Il suffit en effet de rajouter un gestionnaire à l'événement **OnMeasureItem** afin de calculer la hauteur de chaque élément, puis un gestionnaire **OnDrawItem** (très peu différent du précédent) pour dessiner l'élément.

Notez au passage que l'on utilise le fait que la coordonnée verticale d'un trait est celle de son axe central.

Dans un soucis de généricité, nous traçons des traits dont l'épaisseur correspond à la valeur entière du texte situé dans chaque élément de la boîte combo.

```
void __fastcall TFComboTraits::cbEpaisseurMeasureItem(TWinControl *Control,
      int Index, int &Height)
{
  // Conversion du pointeur vers TComboBox*
  TComboBox *cb=dynamic_cast<TComboBox *>(Control);

  // Calcul de la hauteur : 5 fois la valeur de l'élément
  Height=5*(cb->Items->Strings[Index].ToInt());
}
```

Programme 3.3 L'événement OnMeasureItem de calcul de hauteur d'un élément de boîte combo

Commentaire

Ici, le seul paramètre « exotique » est **Height** qui permet de renvoyer la hauteur de l'élément à dessiner.

Le tracé du trait lui même diffère peu de celui de l'exercice précédent et est illustré par le code suivant.

```
void __fastcall TFComboTraits::cbEpaisseurDrawItem(TWinControl *Control,
      int Index, TRect &Rect, TOwnerDrawState State)
{
  // Conversion du pointeur vers TComboBox *
  TComboBox *cb=dynamic_cast<TComboBox *>(Control);
  // On récupère le canevas (contexte d'affichage)
  TCanvas   *cv=cb->Canvas;
  // Coordonnée verticale du trait
  int        vert=(Rect.Top+Rect.Bottom) >> 1;

  // Effacement de la zone par recouvrement avec un rectangle blanc
  cv->Brush->Style=bsSolid;
  cv->Brush->Color=clWhite;
  cv->FillRect(Rect);

  // Tracé du trait avec la bonne épaisseur
  cv->Pen->Style=psSolid;
  cv->Pen->Width=(cb->Items->Strings[Index].ToInt());
  cv->MoveTo(Rect.Left,vert);
  cv->LineTo(Rect.Right,vert);
```

```
}
```

Programme 3.4 tracé d'un trait d'épaisseur variable dans une boîte combo

Solution complète :

La solution se trouve dans l'unité `UComboTraits` du répertoire `Combos` du Power Bear Pack™

3.5.4 Exercice n°4 (✶✶✶)

Etendre la liste de l'exercice n°1 de manière à ce que les 4 derniers éléments entrés soient précédés d'un point noir.

On utilisera la méthode **`TextOut`** de **`TCanvas`** pour afficher un texte.

3.6 Les menus

Il y a grosso modo deux grandes catégories de menus : le menu principal d'une application, qui s'affiche en haut de la fenêtre principale et qui varie finalement très peu au cours de l'exécution et les menus surgissants, activés par le bouton droit de la souris ou la touche spéciale du clavier de Windows qui eux dépendent très fortement du contexte. Ils partagent néanmoins de très nombreuses caractéristiques.

3.6.1 Mise en place d'un menu principal

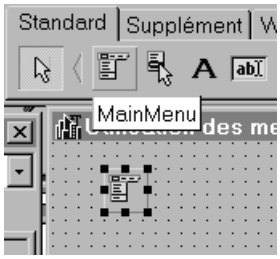


Figure 3.8 Mise en place du menu principal

Tout commence par l'insertion d'un composant non visuel **`TMainMenu`** sur la fiche correspondant à la fenêtre principale de votre application. Ensuite, il suffit de double cliquer sur l'icône de celui-ci ou d'activer « Concepteur de menus » dans son menu contextuel pour entrer dans l'éditeur de menu (commun au menu principal et aux menus surgissants).

Par définition, le menu principal se loge toujours en haut de la fenêtre principal, au dessus des composants possédant l'alignement **`alTop`** s'ils sont présents (des barres d'outils, par exemple). En mode conception, il n'apparaît qu'après la première activation du concepteur de menus.

La propriété principale de chaque composant menu est **`Items`**, propriété regroupant l'ensemble des éléments présents dans le menu, chaque élément étant lui même un objet de la classe **`TMenuItem`**.

3.6.2 L'éditeur de menus

La figure suivante illustre le fonctionnement de l'éditeur de menus. Celui ci permet de construire un menu de façon très intuitive. Comme le montre l'inspecteur d'objets, tout élément de menu est doté d'un identificateur particulier. Ceci a pour conséquence positive qu'il est très facile d'accéder à chacun d'entre eux … et comme aspect malheureux la saturation de l'espace de nommage des identificateurs lorsque les menus regroupent de nombreux éléments. Notons toutefois qu'il est possible, et c'est un cas unique ! de laisser la propriété **Name** en blanc pour un élément auquel l'on n'accèdera pas programmatiquement.

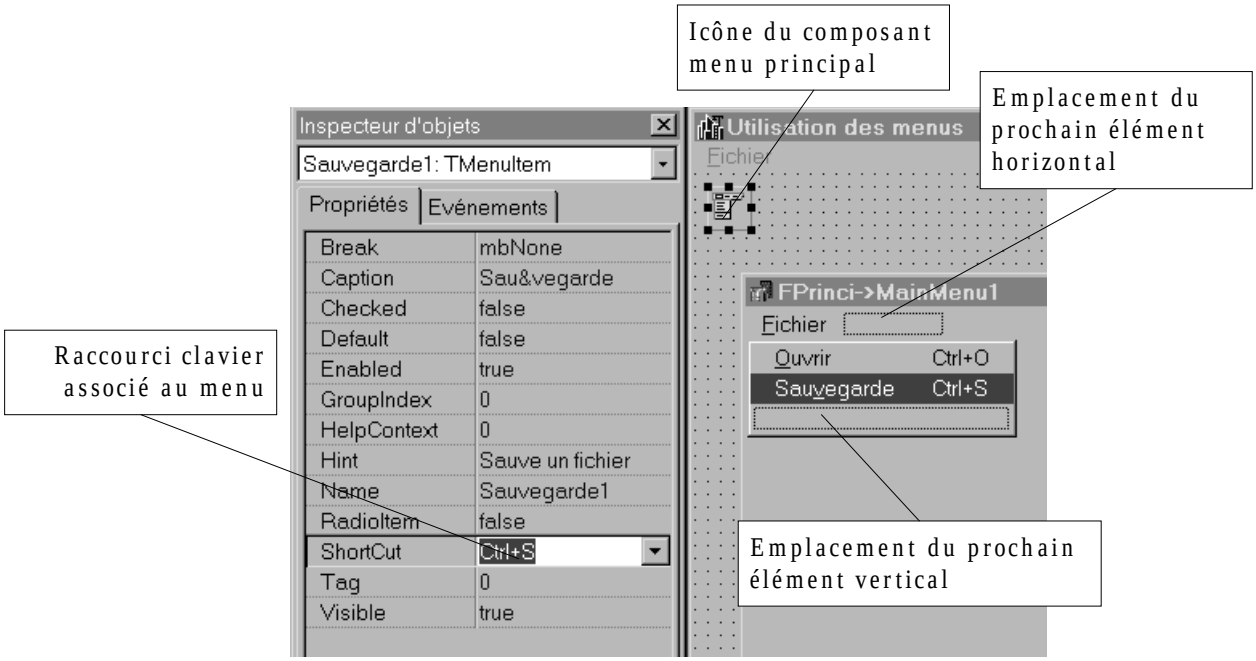


Figure 3.9 Le concepteur (ou éditeur) de menus

Quelques petites indications s'imposent :

- Comme pour les boutons ou les labels de tout poil, il est possible d'utiliser le caractère **&** pour signaler la lettre active d'un élément de menu. Si l'élément en question fait partie de la barre horizontale du menu principal, la combinaison **Alt-lettre-active** permettra d'accéder directement au menu.

- Lorsque vous saisissez le **Caption** d'un élément, le concepteur le valide et en ajoute automatiquement un nouveau du même type. Il ne faut pas s'inquiéter de la présence des ces éléments fantômes présents en mode conception : ils n'apparaîtront pas lors de l'exécution.

- Lorsque l'on désire ajouter un séparateur dans un menu « vertical »,il suffit de taper le caractère tiret – dans son **Caption**.

- « Griser » un menu s'obtient en basculant sa propriété **Enabled** à **false**.

- Positionner à **`false`** la propriété **`Visible`** d'un menu permet de le rendre invisible et donc totalement inactif. Nous verrons que cette propriété est particulièrement utile car la structure d'un menu est figée après la conception : il n'est pas possible d'ajouter ou de supprimer un élément dans un menu lors de l'exécution. Nous pourrons toutefois utiliser la propriété **`Visible`** pour simuler un ajout ou une suppression.

3.6.2.1 L'utilisation de modèles de menus

C++ Builder propose d'utiliser des modèles de menus déjà prêts. Vous pouvez y accéder par le menu contextuel attaché au bouton droit de la souris dans le concepteur de menus. La figure suivante illustre cette possibilité :

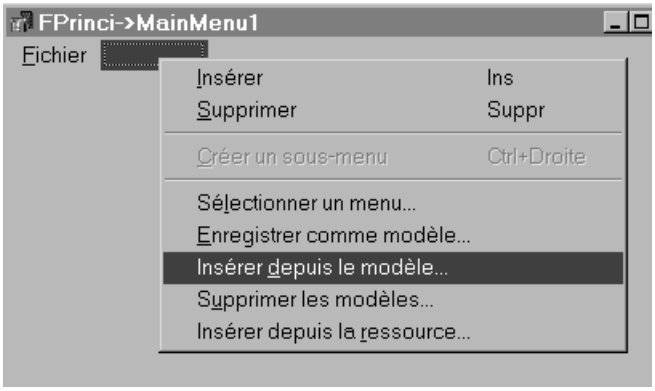


Figure 3.10 Le menu contextuel du concepteur de menus (à répéter très vite 3 fois☺)

Les modèles de menus sont très efficaces car ils reprennent des menus canoniques et très complets. Il vous suffira alors d'ajouter les fonctionnalités qui vous manquent et de supprimer (ou rendre invisibles) celles que vous ne souhaitez par voir apparaître. La figure suivante montre la liste des modèles de menus disponibles par défaut :

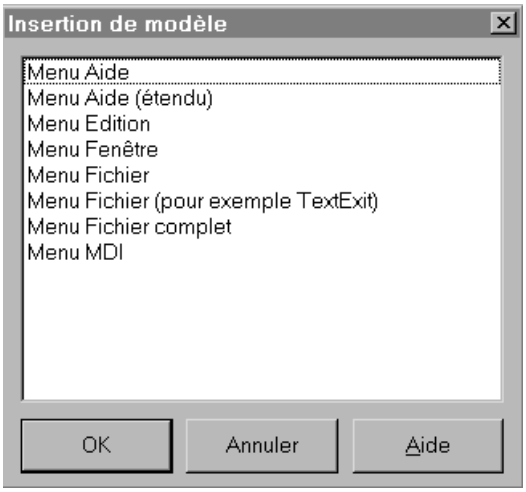


Figure 3.11 Les modèles de menus

Comme vous pouvez le constater, cette liste contient même un menu MDI complet prêt à l'emploi ! La figure suivante montre un menu d'aide directement obtenu après insertion du modèle Aide.

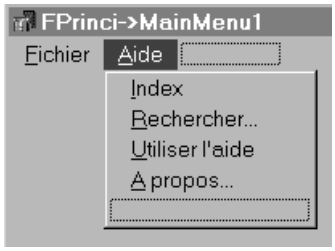


Figure 3.12 Après insertion du menu d'aide par défaut

3.6.2.2 Les sous menus

Il est très simple de créer un sous menu attaché à un élément quelconque : il suffit d'activer l'option correspondante dans le menu contextuel (ou d'activer le raccourci Ctrl-D).

Notez que, dans l'inspecteur d'objets, rien ne permet de différencier un item auquel est attaché un sous menu d'un autre, ce qui peut paraître surprenant car ils ont un comportement tout à fait différent et sont représentés avec une petite flèche vers la droite. La figure suivante montre la création d'un sous menu.

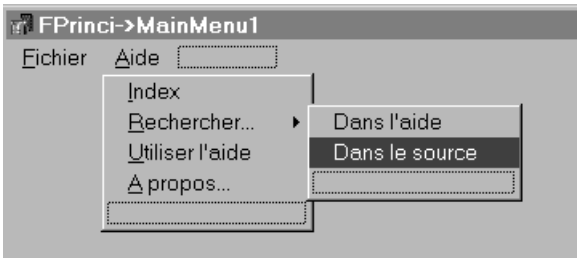


Figure 3.13 Création d'un sous menu

3.6.2.3 Les menus à comportement radio

Il est possible de conférer à certains des éléments d'un menu un comportement rappelant celui des boutons radio dans le sens où seul l'un d'entre eux sera précédé d'un rond noir. Cela permet, par exemple, d'affecter des options.

Pour transformer un ensemble d'éléments en éléments radio, il faut commencer par positionner leur propriété **`RadioItem`** à **`true`** et leur affecter à tous une même valeur de propriété **`GroupIndex`**, laquelle doit être obligatoirement supérieure à 7 afin d'éviter des conflits avec les valeurs prédéfinies de groupes d'éléments de Windows. La propriété **`Checked`** permet de spécifier lequel de ces boutons sera activé par défaut. Attention ! contrairement aux boutons radio, cliquer sur un élément ne l'active pas immédiatement : ceci doit être fait dans le gestionnaire de l'événement **`OnClick`**.

3.6.3 L'utilisation des menus

Les éléments de menu n'ont qu'un seul événement : **`OnClick`** auquel l'on affecte une méthode répondant à l'événement « activation du menu ». Il est nécessaire d'affecter la propriété **`Checked`** d'un élément radio dans le gestionnaire si vous voulez déplacer la marque radio dudit élément.

3.6.4 Un exemple de modification par programmation : la liste des derniers fichiers ouverts

Comme nous l'expliquions quelques lignes auparavant, la structure (nombre des éléments de chaque partie de menu) est figée lors de la conception : la propriété **`Items`** est en lecture seulement. Il n'est donc pas possible d'ajouter ou de retirer un élément du menu en cours d'exécution. Heureusement, il est possible de jouer sur la propriété Visible de chaque élément.

Nous allons profiter de cette aubaine pour construire une liste des 4 derniers fichiers ouverts ayant les propriétés suivantes :

- Lorsque la liste n'est pas vide, elle apparaît en bas du menu fichier, juste avant l'élément Quitter, divisée du reste des éléments par des séparateurs
- Si la liste est vide, elle est totalement invisible
- L'élément le plus ancien est remplacé par le nouveau lorsque la liste est pleine

Afin d'obtenir ce que nous désirons, il est possible d'utiliser la séquence d'instructions :

1. Nous commençons par réserver les emplacements nécessaires comme le montre la figure suivante :

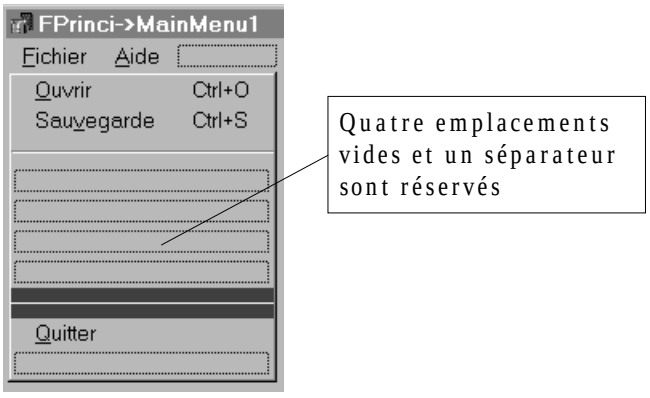


Figure 3.14 Réservation des emplacements de menu pour la liste des derniers fichiers ouverts

2. Dans un second temps, nous basculons la propriété **`Visible`** des 4 emplacements et d'un des séparateurs à **`false`**. Attention ! si vous ne spécifiez pas de **`Caption`** pour un élément, par défaut son nom restera en

blanc, effet dont nous ne voulons pas. Aussi, soit vous donnez un **Caption** qui sera de toute façon remplacé par la bonne valeur au moment de l'affichage, soit vous fournissez un nom au composant.

3. Afin de pouvoir accéder facilement aux quatre emplacements, nous ajoutons dans les déclarations privées de la classe Fiche :

   - un tableau de 4 **TMenuItem** qui sera affecté aux adresses des 4 éléments dans le constructeur

   - une variable entière indiquant le nombre d'éléments présents dans la liste

4. Il ne reste plus qu'à rendre visibles les éléments et à leur affecter le bon **Caption**. L'exemple qui suit remplit tout simplement cette liste grâce à un bouton et une zone d'édition.

```
…
    // Variables affectées individuellement aux éléments de menu
    // destinés à la liste des derniers fichiers ouverts
    TMenuItem *MenuFich1;
    TMenuItem *MenuFich2;
    TMenuItem *MenuFich3;
    TMenuItem *MenuFich4;
private:    // Déclarations de l'utilisateur
    // Déclaration d'une constante indiquant le nombre d'éléments de la
liste
    enum {NB_FICHIERS=4};
    // Tableau permettant de manipuler facilement les éléments de menu
    // destinés à la liste des derniers fichiers ouverts
    TMenuItem *MenuFichiers[NB_FICHIERS];
    // Variable comptant le nombre d'éléments présents
    int        nbFichiers;
…
```

Programme 3.5 Déclarations relatives à la mise en place de la liste des derniers fichiers ouverts

```
__fastcall TFPrinci::TFPrinci(TComponent* Owner)
    : TForm(Owner)
{
  // Remplissage (bourrin) du tableau des éléments
  MenuFichiers[0]=MenuFich1;
  MenuFichiers[1]=MenuFich2;
  MenuFichiers[2]=MenuFich3;
  MenuFichiers[3]=MenuFich4;
  nbFichiers=0;
}
//---------------------------------------------------------------------
---
void __fastcall TFPrinci::bnElementClick(TObject *Sender)
{
  // Affichage du séparateur
  N2->Visible=true;
  // Mise en place du Caption et affichage de l'élément
  MenuFichiers[nbFichiers]->Caption=editElement->Text;
  MenuFichiers[nbFichiers]->Visible=true;
  // Mise en place du cyclage
  nbFichiers++;
```

```
  nbFichiers%=NB_FICHIERS;
}
```

Programme 3.6 Code d'implémentation du remplissage de la liste

Comme souvent, ce code répète plusieurs fois la même opération (mettre la propriété **`Visible`** des éléments à **`true`**) car cela coûterait aussi cher d'effectuer le test de visibilité.

3.6.5 Les menus surgissants

Les menus surgissants (*popup menus*) sont en tout point semblables (du moins pour ce qui est de leur conception) au menu principal d'application. La seule différence réside dans leur activation qui est contrôlée par deux propriétés.

`AutoPopup` est une propriété de **`TPopupMenu`** qui indique si ce menu est activable par un clic sur le bouton droit de la souris (ou l'enfoncement de la touche idoine du clavier) ou s'il est uniquement accessible par programmation (méthode **`Popup`**). Cette propriété est par défaut à **`true`**, ce qui correspond bien à l'utilisation la plus fréquente de ces menus.

`PopupMenu` est une propriété définie par tous les contrôles et qui indique l'identificateur du menu surgissant actif dans ce contrôle. Ainsi, il est possible de spécifier un menu surgissant par contrôle. Si cette propriété n'est pas renseignée, alors le menu surgissant est celui du parent.

3.7 Les boîtes déroulantes

Le fonctionnement des ascenseurs est quelque peu étrange en C++ Builder. Nous allons donner un aperçu des possibilités qu'ils offrent.

3.7.1 Généralités

Les boîtes déroulantes **`TScrollBox`** (sur la palette Supplément) permettent de faire défiler des composants à l'aide d'ascenseurs. Pour bien comprendre leur fonctionnement, il nous faut définir un peu de vocabulaire à l'aide de la figure suivante :

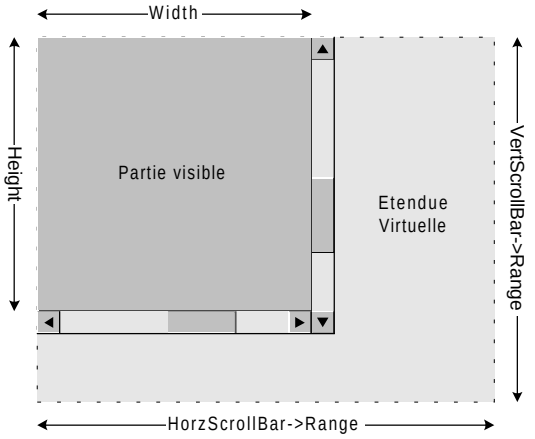


Figure 3.15 Partie visible et étendue virtuelle d'un `TScrollBox`

La partie « visible » d'un composant **TScrollBox** est spécifiée, comme à l'accoutumée par ses propriétés **Width** et **Height**. Le nouveau concept est celui d'étendue virtuelle. En effet, le défilement sert à afficher les parties cachées de l'étendue virtuelle du composant. La largeur et la hauteur de cette dernière sont respectivement indiquées par les propriétés **Range** des ascenseurs horizontaux et verticaux lesquels sont stockés dans les propriétés **HorzScrollBar** et **VertScrollBar** de la **TScrollBox**.

L'étendue virtuelle peut varier au cours de la vie de l'application. Lorsqu'elle devient inférieure ou égale à la partie visible, les ascenseurs sont cachés.

Autre chose importante à savoir : modifier la position des ascenseurs ne génère pas d'évènement particulier. Vous ne pouvez donc pas savoir en temps réels si leur position a changé. En conséquence, vous devez prévoir l'affichage complet de la zone virtuelle.

3.7.2 Que peut on mettre dans une TScrollBox ?

Potentiellement, il est possible d'intégrer dans une **TScrollBox** n'importe quel type de contrôle, ce qui peut s'avérer pratique si l'on doit créer une interface graphique à la main en fonction d'un fichier de configuration.

Supposons par exemple, que vous écriviez une application dont certaines options booléennes sont passées dans un fichier. Vous souhaitez associer une case à cocher à chacune de ces options. Comme vous ne connaissez pas leur nombre à l'avance, vous êtes obligés de les construire à la main lors de l'exécution de votre programme.

Afin de ne pas surdimensionner votre fenêtre, il est possible de les placer dans une **TScrollBox**. Si vous activez la propriété **AutoScroll**, l'espace virtuel sera étendu automatiquement lorsque vous ajouterez des contrôles dans votre boîte déroulante. De toute façon, il vous est toujours possible de modifier les **Range** à votre guise.

Le plus souvent, les **TScrollBox** sont associées à un contrôle graphique ou un contrôle fenêtré unique dans le but d'afficher un document de grande taille. C'est ce que l'on appelle une vue.

3.7.3 Exercice résolu : affichage d'un dessin avec facteur de zoom (✶✶✶)

L'exemple que nous allons traiter utilise un **TImage** placé dans une **TScrollBox** pour afficher un dessin de taille variable en fonction d'un certain facteur de zoom. Afin de ne pas compliquer inutilement l'exercice, nous afficherons une suite de 25 cercles concentriques.

Le facteur de zoom (variant entre 1 et 20) est indiqué par un composant **TTrackBar** lequel modélise un curseur placé le long d'une règle. Les propriétés les plus intéressantes sont :

Position	Valeur du curseur
Min	Valeur minimale du curseur, associée à la position « tout à gauche »
Max	Valeur maximale du curseur, associée à la position « tout à droite »
Frequency	Nombre d'unités séparant deux graduations sur la règle

Tableau 3.2 Quelques propriétés du contrôle `TTrackBar`

On démarre avec une taille d'image de 50x50 correspondant au facteur de zoom=1. A chaque fois que le facteur de zoom change, vous modifierez la taille du **`TImage`**, (simplement en multipliant la taille de base par le facteur de zoom), modifierez l'étendue virtuelle de la **`TScrollBox`** si cela est nécessaire et mettrez à jour l'affichage des cercles concentriques.

Pour bien exécuter cet exercice, il est préférable d'instancier manuellement le composant **`TImage`** d'affichage du dessin. En effet, lorsque l'on modifie les propriétés **`Width`** et **`Height`** d'un **`TImage`**, cela n'a pas de répercussion directe sur la taille de la zone cliente (**`ClipRect`**), laquelle est dans une propriété en lecture seulement. En effet, cela reviendrait à réallouer la zone de tampon mémoire associée à la **`TImage`**.

L'interface à créer ressemble à :

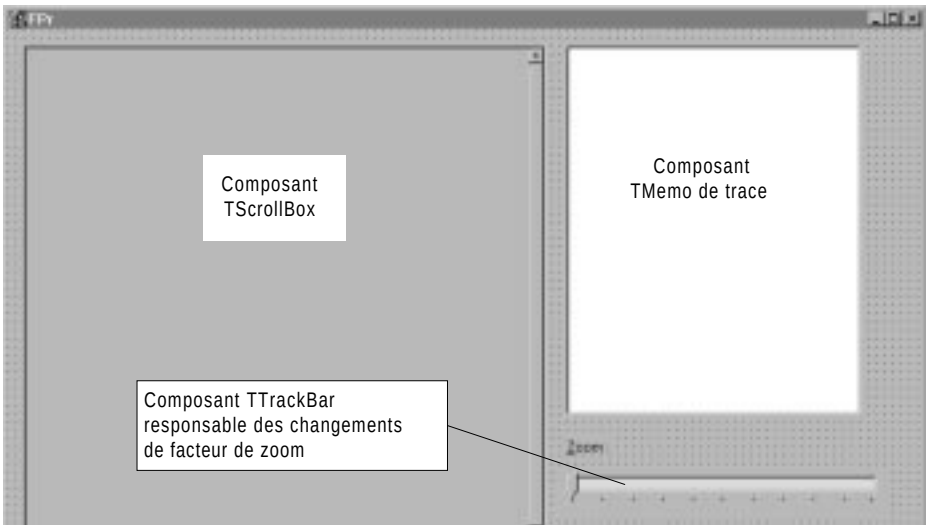


Figure 3.16 Utilisation d'une `TScrollBox` **pour le problème du dessin à Zoomer**

Vous pouvez suivre les instructions suivantes :

- Détruire le composant **`TImage`** en place (**`delete`**)
- Créer un composant **`TImage`**, le seul paramètre à passer au constructeur est le propriétaire, c'est à dire la **`TScrollBox`**.
- Modifier les propriétés fondamentales de la **`TImage`** :

- **Parent** : identificateur de la **TScrollBox**, gère l'axe Z
- **Width** et **Height** : taille de l'image à afficher
- **Top** et **Left** : positionnés à zéro

- Afficher le dessin

Cette technique, au demeurant fort simple souffre d'un handicap colossal : il faut allouer un **TImage** capable de contenir tout le dessin ce qui demande des ressources mémoire conséquentes. Aussi, pour des documents de grande taille, il faudra recourir à une autre technique qui elle n'utilise plus des **TScrollBox** mais des **TScrollBar**.

Pour aller plus loin dans l'exercice (✶✶) :

- Centrez l'image dans la **TScrollBox** si elle est plus petite que la taille statique de la **TScrollBox**,
- Ajoutez un composant zone d'édition affichant en permanence le facteur de zoom et permettant de le saisir. On effectuera tous les contrôles d'erreur nécessaires (valeurs en dehors de [Min, Max], saisie de valeur non numérique, etc.)
- Essayez d'atteindre les limites d'allocation de C++ Builder et fixez la valeur supérieure du zoom en conséquence.

3.8 Les ascenseurs simples

3.8.1 Généralités

Les ascenseurs simples sont encapsulés dans le composant **TScrollBar** (palette standard). Une valeur numérique indiquant la position de leur « curseur » au sein de leur « étendue » leur est associée :

Leurs propriétés fondamentales sont les suivantes :

Min	Valeur minimale de l'étendue du curseur
Max	Valeur maximale de l'étendue du curseur
Position	Valeur courante du curseur
Kind	Spécifie si l'ascenseur est horizontal (valeur **sbHorizontal**) ou vertical (**sbVertical**)
SmallChange	Valeur d'incrémentation du curseur lorsque l'utilisateur utilise les flèches.
LargeChange	Valeur d'incrémentation du curseur lorsque l'utilisateur clique dans la barre d'ascenseur ou utilise PgUp ou PgDown

Tableau 3.3 Propriétés fondamentales des ascenceurs (`TScrollBar`)

Contrairement aux ascenseurs des **`TScrollBox`**, les **`TScrollBars`** génèrent un évènements lorsque l'on modifie leur valeur, il s'agit de **`OnChange`**.

3.8.2 Exercice (✶✶)

Enoncé :

Reprendre l'exercice précédent mais en réalisant l'affichage dans un **`TImage`** fixe et en faisant défiler l'image avec des ascenseurs simples.

Solution partielle :

Il est relativement simple d'adapter le code de l'exercice précédent à celui-ci. Aussi, je vous conseille de dupliquer tout le code du précédent projet dans un nouveau répertoire et de modifier l'application existante.

Première chose : supprimer la **`TScrollBox`** et ajouter :

- Un composant **`TImage`** que l'on aura pris soin de poser sur un **`TPanel`**
- Deux composants **`TScrollBar`**, un horizontal et un vertical situés de part et d'autre du **`TImage`**

L'interface obtenue est montrée par la figure suivante :

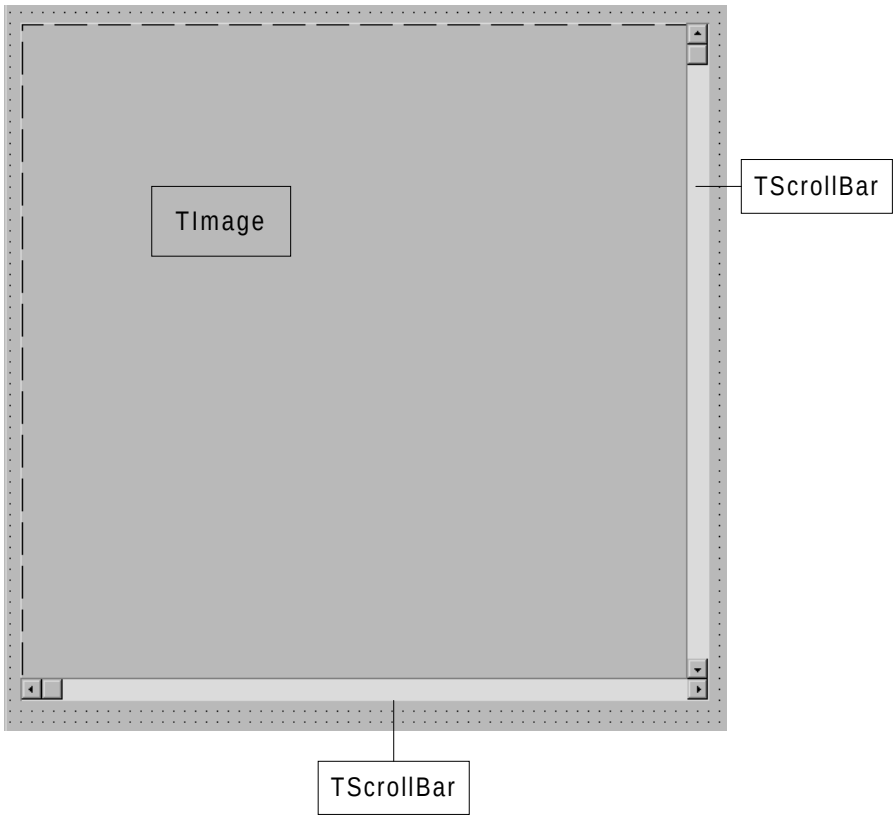


Figure 3.17 Utilisation de `TScrollBar`

La gestion du zoom et du défilement est totalement différente dans le sens où, d'un côté, on construit complètement l'image un document et les barres de défilement ne font que faire glisser une loupe dessus alors que de l'autre côté, c'est le programmeur qui gère le scroll en récupérant la position des barres de défilement pour construire l'image de la partie visible d'un document.

Cette solution est un peu plus compliquée à mettre en œuvre mais beaucoup plus souple au sens où elle autorise des documents d'une taille de visualisation virtuellement illimitée.

En effet, dans le cas précédent, on utilise un composant **`TImage`** dans lequel toute la scène est dessinée. Aussi, la taille de se composant croit elle avec le facteur de Zoom. Lorsque l'on utilise les barres de défilement simples, la taille du composant **`TImage`** ne varie pas.

3.9 Les barres d'outils de C++ Builder

La création d'une telle barre d'outils se fait en utilisant la classe **`TToolBar`**. Par défaut, un objet de classe **`TToolBar`** se place en haut de sa fenêtre parent. C'est un objet conteneur destiné à recevoir divers objets fils parmi lesquels on recense :

- N'importe quel type de contrôle
- Des séparateurs
- Des boutons gadgets

Ces deux dernières classes étant plus spécifiquement liées aux barres de contrôles ou autres palettes flottantes.

3.9.1 Insertion de contrôles standard dans une barre d'outils

Si les contrôles n'existent pas encore, il suffit de sélectionner la barre de contrôle avant de les placer, il s'insèreront directement à leur place. On remarque, que, par défaut, les contrôles sont collés les uns aux autres constituant des groupes. Heureusement, il est possible de laisse de l'espace en insérant des séparateurs avec une commande de menu rapide (bouton de droite). Ceux-ci sont aisément repositionnables et redimensionnables. La présence de séparateurs constitue des groupes de contrôles.

En revanche, si les contrôles existent déjà, il faut les déplacer par des opérations Couper / Coller, le collage devant s'effectuer barre d'outils sélectionnée.

3.9.2 Les boutons gadgets

Les boutons gadgets ressemblent à s'y méprendre aux anciens turbo boutons de Delphi 2. Toutefois, il ne faut pas s'y tromper, ce ne sont pas des contrôles car ils ne reposent pas sur une fenêtre. Ils ne peuvent donc pas recevoir et émettre des messages Windows de leur propre chef mais doivent être utilisés au sein d'une fenêtre dérivant de **`TGadgetWindow`**, comme, par exemple, **`TToolBar`**.

C++ Builder masque le travail de la **GadgetWindow** en proposant au développeur des gestionnaires d'événements au niveau de la fiche et qui semblent directement liés aux boutons gadgets.

Ils sont créés en utilisant la commande **New Button** du menu rapide des objets de classe **TToolBar**. A l'instar des turbo boutons (qui étaient présents dans Delphi 2 pour les émuler) ils peuvent être de type commande ou *settings* et adopter un comportement radio dans un groupe. Ici un groupe est un ensemble contigu de boutons gadgets entouré de séparateurs.

Les différents bitmaps associés aux boutons présents dans une barre d'outils sont tous regroupés au sein d'une propriété de la barre d'outils : **ImageList**. Chaque bitmap est ensuite adressé dans cette propriété via son index.

Voici donc la marche à suivre pour créer une barre d'outils avec des boutons gadgets :

1. Créer tous les bitmaps en les enregistrant dans des fichiers .**bmp**. Vous disposez pour cela, soit de l'éditeur intégré, soit de tout utilitaire capable de générer des .**bmp**.
2. Rassembler tous les bitmaps dans un objet **TImageList** (on peut alors détruire les .**bmp** ...)
3. Créer l'objet **TToolBar** et lui associer la **TImageList**
4. Créer les boutons gadgets, leur adjoindre un style et leur associer leur bitmap

4. Utilisation minimaliste des bases de données avec C++ Builder

Ce document n'a d'autre prétention que de présenter sommairement et par ordre chronologique les fonctionnalités minimales permettant de travailler avec des bases de données sous C++ Builder. Celles-ci sont gérées par une couche logicielle intitulée BDE pour Borland **`Database Engine`**. Ce moteur de bases de données est très souple (car acceptant de nombreuses sources de données) et remplace l'ancien **`Paradox Engine`**. Il est compatible avec les langages d'interrogation SQL ANSI et QBE.

Nous allons réaliser un exercice sur une base de données traitant d'ouvrages de science fiction.

4.1 Mise en place de la base de données

La première étape consiste à créer une base de données par l'intermédiaire d'un alias de base de données.

Un *alias de base de données* est un identificateur *unique* permettant d'accéder à une base de données exploitable avec C++ Builder. Celle-ci peut être de plusieurs types en fonction des drivers installés sur votre système :

- Tout lien ODBC reconnu par votre environnement Windows (toujours présent)
- Une base de données Access (option)
- Une base de données SQL au format Interbase (option souvent présente)
- Un lien SQL vers un serveur Oracle, Sybase, DB2, SQL Server etc. (option).
- Un répertoire regroupant des fichiers DBASE ou PARADOX (option par défaut, donc toujours présente ☺)

La création d'un alias se fait très simplement avec un outil nommé soit « Administrateur BDE »

Dans le cas qui nous intéresse, nous créons un alias pour l'ensemble de fichiers au format Paradox situés dans le répertoire `c:\f2\Builder\donnees`, alias que nous nommerons `TPF2ZZ2`.

4.1.1 L'outil « Administrateur BDE »

L'administrateur BDE est un outil à la fois très simple et très complet. Il est capable d'afficher soit l'ensemble des alias déjà présents (sélectionnez l'onglet « Bases de données) soit des informations concernant le système telles que l'ensemble des formats de bases de données disponibles ou le format des nombres (sélectionnez l'onglet « Configuration ».

La fenêtre principale de l'administrateur BDE se décompose en deux volets :

- Dans le volet de gauche, vous sélectionnez un objet (un alias de bases de données en mode « Bases de données)
- Dans le volet de droite, vous obtenez la liste de ses caractéristiques.

La figure suivante montre l'aspect typique de cet outil en mode « Configuration ». La sélection porte sur un pilote de bases de données natif.

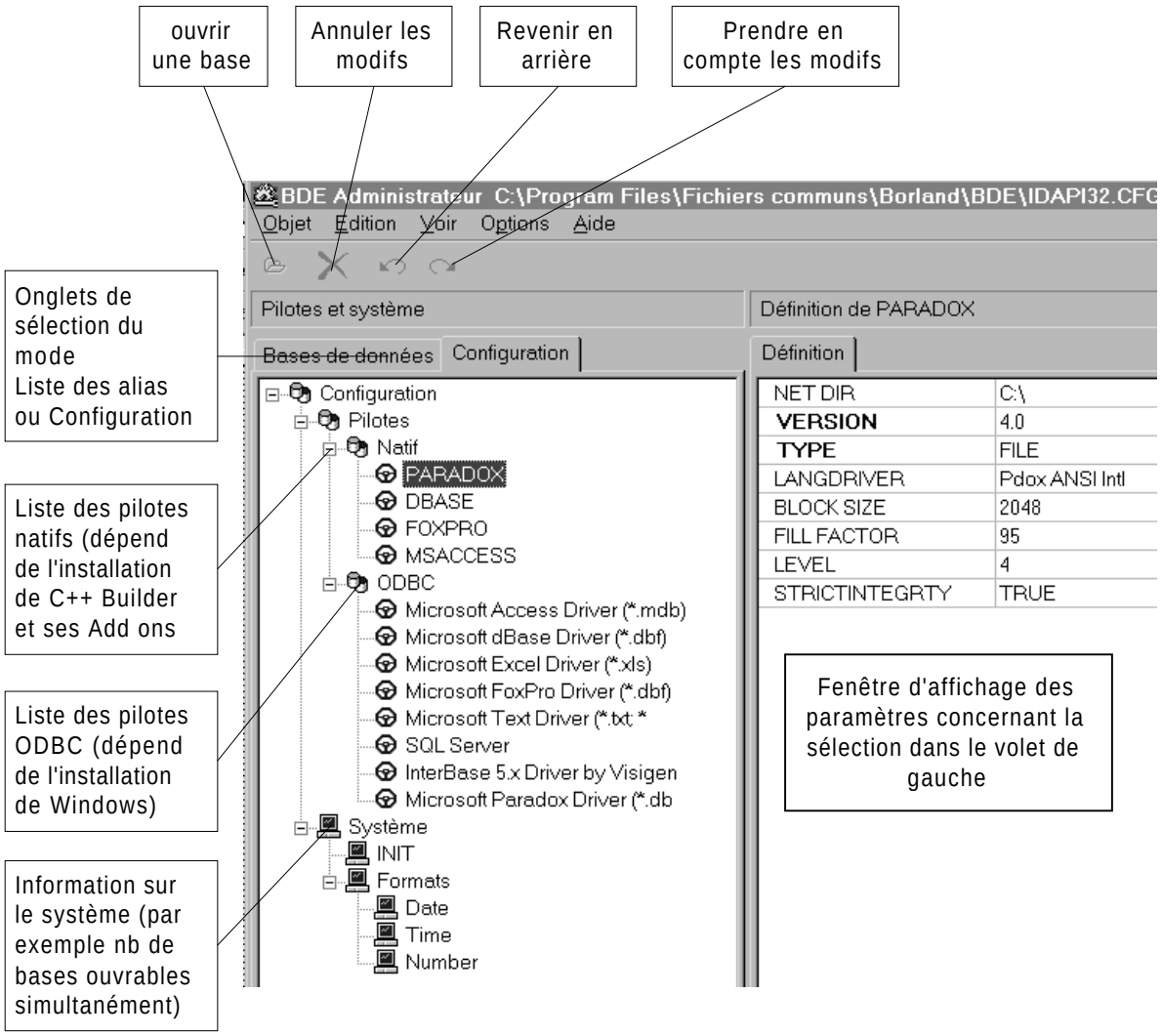


Figure 4.1 L'administrateur BDE en mode configuration

Contrairement à ce que l'on pourrait penser, les performances d'un pilote natif sont toujours supérieures à celles d'un pont ODBC. En effet, certains ponts ODBC sont d'excellentes factures alors que l'on trouve des pilotes exécrables. Aussi, il est préférable d'effectuer quelques tests.

4.1.2 Création de l'alias

La démarche à suivre est la suivante :

- Lancement de l'utilitaire Administrateur BDE

- Sélection de l'onglet « Bases de données » Au passage, consultez les informations concernant quelques alias afin de vous familiariser avec leur format.

- Pour créer un nouvel alias, sélectionnez Objet→ Nouveau dans le menu

- Il faut ensuite sélectionner le type de bases de données. Pour notre exemple, choisissez STANDARD, ce qui correspond à un ensemble de fichiers DBASE ou Paradox.

  Cette opération est illustrée sur la figure suivante :

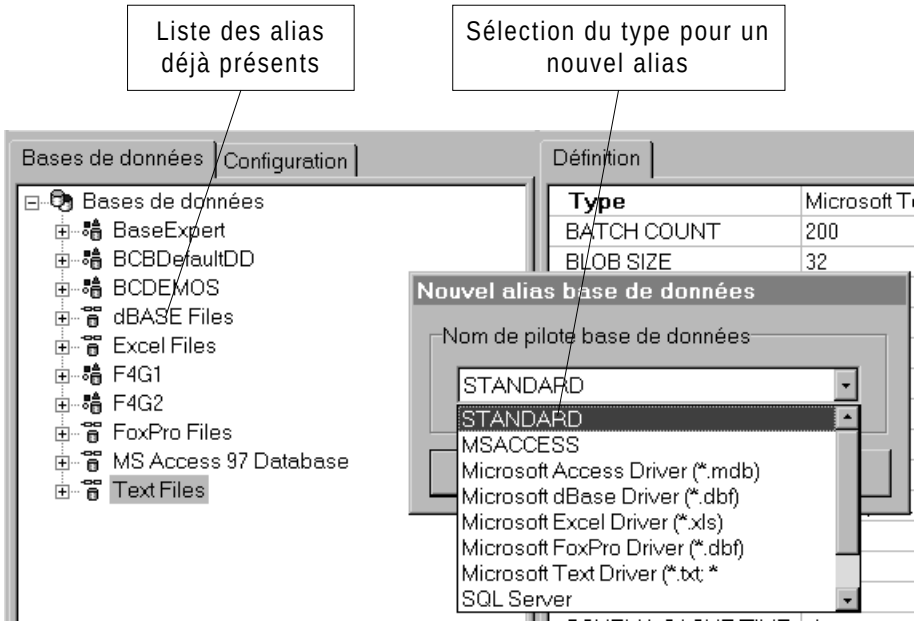


Figure 4.2 Sélection du type de la base de données

- Une fois le type choisi, il vous reste à remplir diverses rubriques concernant votre nouvelle base, comme le montre la figure suivante :

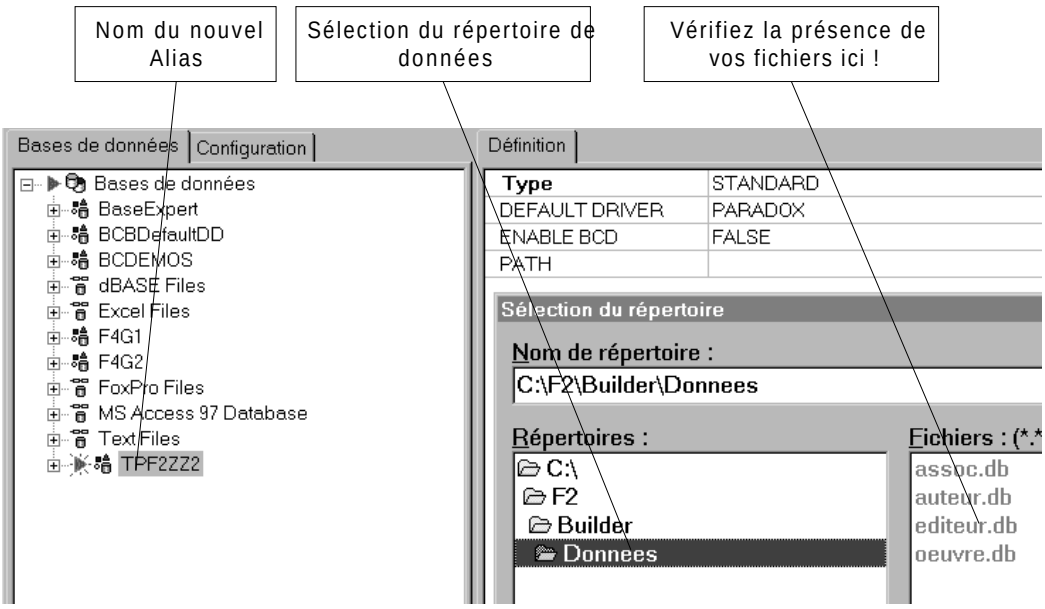


Figure 4.3 Sélection du nom de l'alias et du chemin des données

- Vous devez renseigner absolument :
  - Le nom de l'alias (le nom par défaut est épouvantable ☺)
  - Le chemin des données
- En option, vous pouvez activer le traitement des nombres au format BCD qui autorise une précision infinie au détriment des performances.
- Finalement, il vous reste à valider en activant la flèche bleue. Une boîte de dialogue vous demande alors de confirmer. A partir de ce moment là, votre base de données est accessible via C++ Builder.

Notons également qu'un utilitaire nommé « Module de base de données » décrit dans un chapitre ultérieur permet de créer des fichiers de données au format PARADOX ou DBASE ainsi que de tester des requêtes SQL ou QBE simple.

Bien que rudimentaire, cet outil permet de créer simplement des tables et de leur associer des clefs primaires ou secondaires, des fichiers d'index secondaires ou des contraintes d'intégrité.

4.2 Accès aux données dans C++ Builder

La première chose à faire est, bien entendu, de créer un projet C++ Builder. Ensuite, il faut savoir que C++ Builder effectue une ségrégation forte entre les variables qui représentent les données en tant que *document* et les éléments visuels (ou *d'interface*) qui permettent à l'utilisateur de les manipuler. C'est encore un bel exemple d'application du modèle Document / Visualisation.

Il est préférable de toujours regrouper les éléments d'accès aux données – qui sont des composants non visuels – dans une unité spécialisée connue sous le nom de Module de Données. Celui-ci est créé par la commande « Fichier → nouveau module de données » et se manipule ensuite comme une fiche / unité standard.

La base choisie comme exemple regroupe trois relations :

- **« `Oeuvre.db` »** regroupe les données relative à des ouvrages de Science-fiction. Pour chaque livre, nous avons les champs suivants : l'index dans la base (Clef primaire), le titre, une référence à un auteur (clef étrangère) et une référence à un éditeur (clef étrangère).
- **« `auteur.db` »** contient les données décrivant les auteurs, à savoir un index (clef primaire) et le nom
- **« `editeur.db` »** a la même structure qu'auteurs mais décrit, comme son nom l'indique, les éditeurs d'ouvrages de science fiction (collections de poche)
- Dans un second temps, nous utiliserons une quatrième base nommée **« `assoc.db` »** qui permettra d'associer plusieurs auteurs à une même œuvre.

La première chose à faire consiste à associer un composant **TTable** à chacune de ces relations. Pour ceci, nous choisissons un composant **TTable** dans la palette « Accès aux données » que nous collons dans le module de données. L'inspecteur d'objets permet ensuite de l'associer à un alias de base de données (champ **Database name**) puis une relation (**Table name**). Il est recommandé de donner à ce composant un nom en rapport avec celui de la relation. Dans cet exemple, nous choisirons **TableOeuvres**, **TableAuteurs** et **TableEditeurs**.

A ce moment, il est possible d'ouvrir la table en armant le flag **active** dans l'inspecteur de données. L'index utilisé est basé sur la clef primaire sauf si l'utilisateur active un index secondaire en modifiant (par l'intermédiaire d'une boîte combo présentant tous les index secondaires définis sur la relation) la propriété **IndexName**.

La figure suivante illustre un composant **TTable** placé sur un module de données.

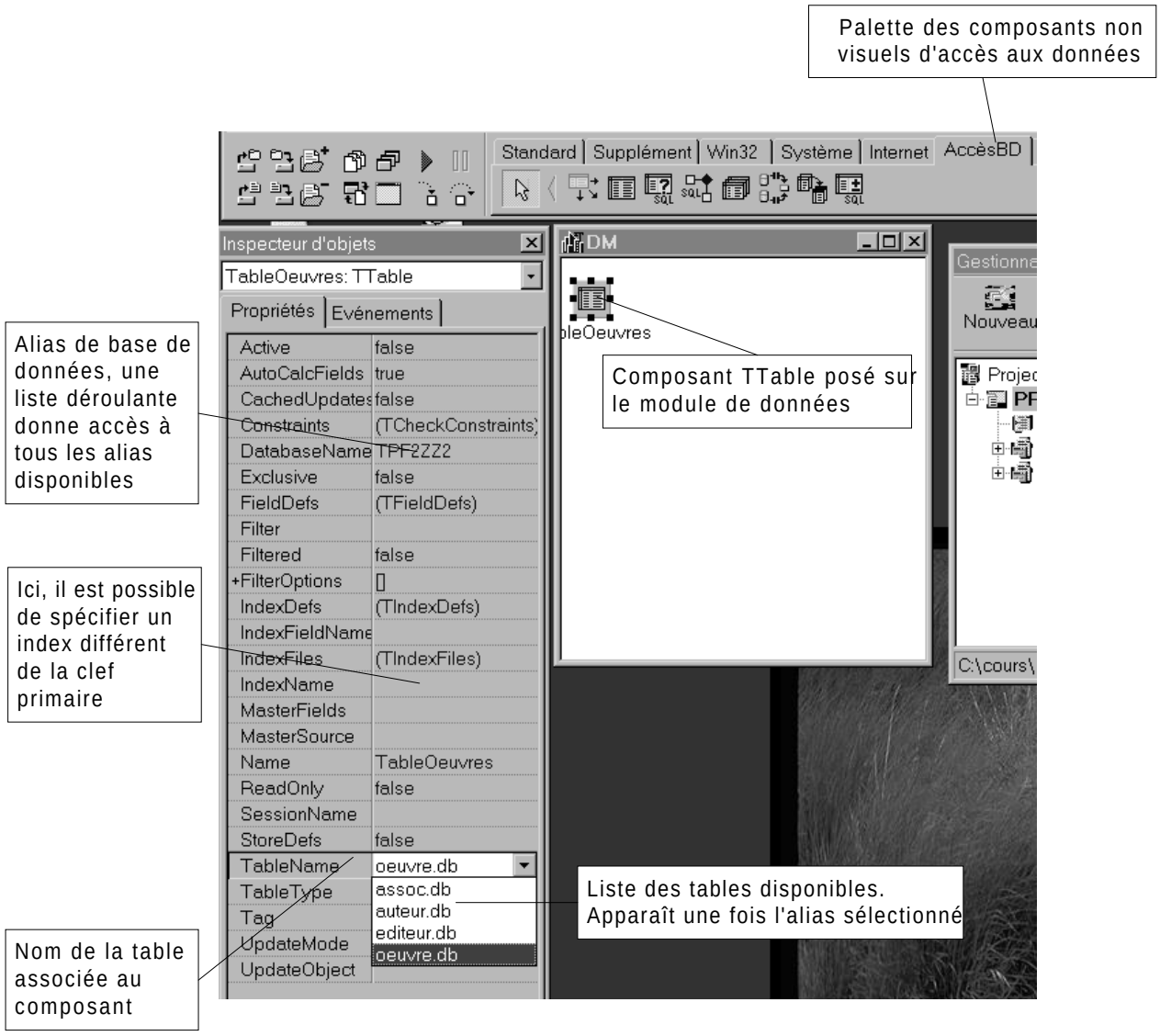


Figure 4.4 Première étape de la création d'un composant TTable : sélection d'un alias de base de données

Les principales propriétés de **TTable** sont les suivants :

- **DatabaseName** : nom de l'alias de base de données ou du composant **TDatabase** spécifiant la base de données au sein de laquelle nous allons pêcher une table. Cette propriété est commune à **TTable**, **TQuery** et **TStoredProc**.

- **TableName** : nom de la relation représentée par le composant. Une fois la propriété **DatabaseName** renseignée, vous pouvez choisir le nom de la table dans une liste déroulante.

- **IndexName** : nom de l'index actif sur la table. Par défaut, c'est-à-dire si le champ est vide, il s'agit de la clef primaire. Vous avez la possibilité de sélectionner un index dans une liste déroulante une fois la propriété **TableName** fixée.

- **Active** : ce champ indique si la table est ouverte ou non. Vous ne pouvez le basculer à **true** qu'une fois les champs **DatabaseName** et **TableName** renseignés.

  En outre, Il faut savoir que toute modification sur la structure de la table entraîne irrémédiablement le passage de la propriété à **false**.

- **FieldDefs** contient la définition de chacun des champs de données de la table.

Il est possible d'inclure d'autres composants non visuels dans un module de données. Nous citerons en particulier :

- Des composants représentatifs d'une base de données (**TDatabase**). Ceux-ci permettent d'intercaler une couche supplémentaire entre les tables (ou les requêtes) et les alias de bases de données. En effet, si vous disposez de plusieurs bases partageant la même structure et que vous souhaitez pouvoir utiliser votre code avec l'une ou l'autre indifféremment, il est possible d'utiliser une variable **TDatabase** qui pointera sur le bon alias. Les **TTable** ou **TQuery** peuvent alors référencer ce composant dans leur propriété **DatabaseName**.

- Des requêtes SQL (**TQuery**). Les requêtes et les tables ont un ancêtre commun nommé **TDataSet** dans le sens ou ces deux composant ont pour résultat des ensembles de données que l'on pourra afficher. Les requête SQL de C++ Builder sont paramétrées et il est possible de récupérer la valeur de ces paramètres directement dans certains composants ou par jointure sur une table ou le résultat d'une autre requête.

- Des procédures stockées SQL (**TStoredProc**) c'est à dire du code SQL précompilé pour une exécution plus rapide. Il s'agit habituellement de code de maintenance et non pas de code d'extraction.

- Des sources de données de type **`TDataSource`**. Ces composants (non visuels) établissent le lien entre les ensembles de données (Tables ou Requêtes) *i.e.* la partie Document du modèle C++ Builder avec les composants visuels chargés de permettre la manipulation des bases de données *i.e.* la partie Visualisation du modèle C++ Builder.

Bien que ce ne soit pas requis, il est préférable de placer les **`TDataSource`** (ou sources de données) dans le module de données. Personnellement, je recommande de les placer à la droite du composant **`TDataSet`** sur lequel ils travaillent.

Il est possible d'avoir plusieurs sources de données pour le même ensemble de données. On dispose ainsi de plusieurs visualisations sur un même document. Le placement des objets sur le module de données n'ayant pas d'incidence sur le code produit, il vous appartient de la structurer de la manière la plus cohérente possible.

La propriété la plus importante des composants **`TDataSource`** est **`DataSet`**. En effet celle-ci indique sur quel ensemble travaille la source de données. Vous pouvez spécifier dans cette propriété tout composant héritant de **`TDataSet`**, c'est à dire (pour ce qui concerne les composants de base) **`TTable`** et **`TQuery`**.

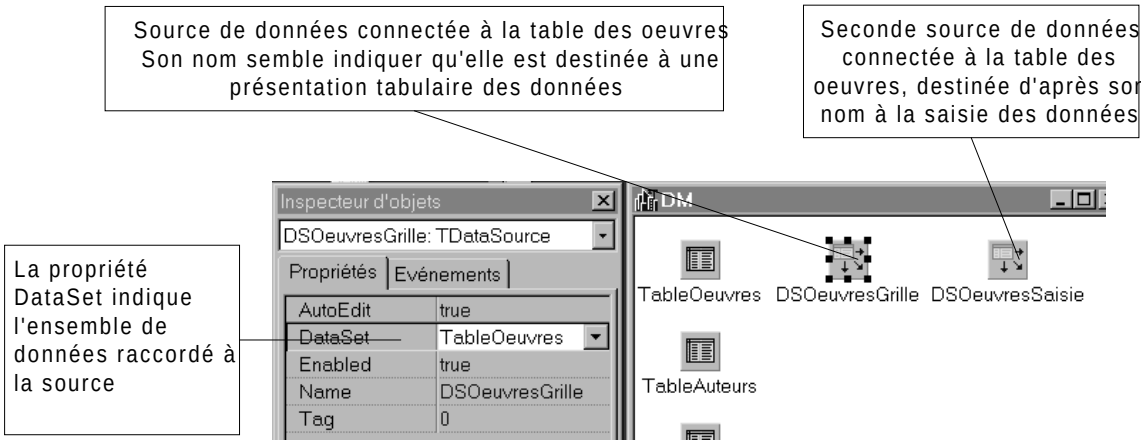


Figure 4.5 Les composants `TDataSource`

4.3 Les contrôles orientés bases de données

Ce sont des versions spécialisées dans l'édition des bases de données des contrôles standard de Windows ou d'OWL. On retrouve ainsi, par exemple, des boîtes d'édition, des listes ou des boutons radio orientés bases de données. Le contrôle le plus important est néanmoins celui qui permet de présenter une relation ou le résultat d'une requête sous forme tabulaire : **`TDBGrid`**. Tous ces composants se trouvent dans la palette « Contrôle BD ».

Les contrôles orientés bases de données se placent sur les fiches et non pas sur le module de données : ils constituent la partie interface du modèle Document / Visualisation de C++ Builder.

Tous sont reliés aux composants non visuels via les sources de données. Afin d'accéder à ces dernières, il ne faut pas oublier d'inclure l'entête du module de bases de données dans la ficher de présentation (Fichier → Inclure l'entête d'unité).

4.3.1 Présentation tabulaire d'une table ou d'une requête

Nous donnons ici l'exemple de présentation d'une table mais ce qui suit s'applique à tout composant de type **TDataSet**, une requête par exemple. Le composant **TDBGrid** présente les données sous leur forme la plus naturelle celle d'un tableau où chaque colonne correspond à un champ et chaque ligne à un tuple.

Voici la marche à suivre pour créer une représentation d'un **TDataSet** :

- Placer un composant **TDBGrid** sur la fiche.
- Associer à sa propriété **DataSource** une source de données présente dans le module de données (il sera sans doute nécessaire d'inclure le fichier d'entête créer un lien uses depuis le module de données vers la fiche de présentation à l'aide du menu « Fichier → Inclure l'entête d'unité »).

Aussitôt cette dernière opération réalisée, la grille se remplit des données présentes dans la relation. Si rien ne se passe, il faut vérifier la propriété **Active** de la relation que l'on souhaite présenter. Le tableau créé compte autant de colonnes que de champs dans la relation et autant de lignes que de tuples. Chaque colonne possède un titre dont la valeur est prédéfinie au titre du champ. La figure suivante illustre le composant **TDBGrid**.

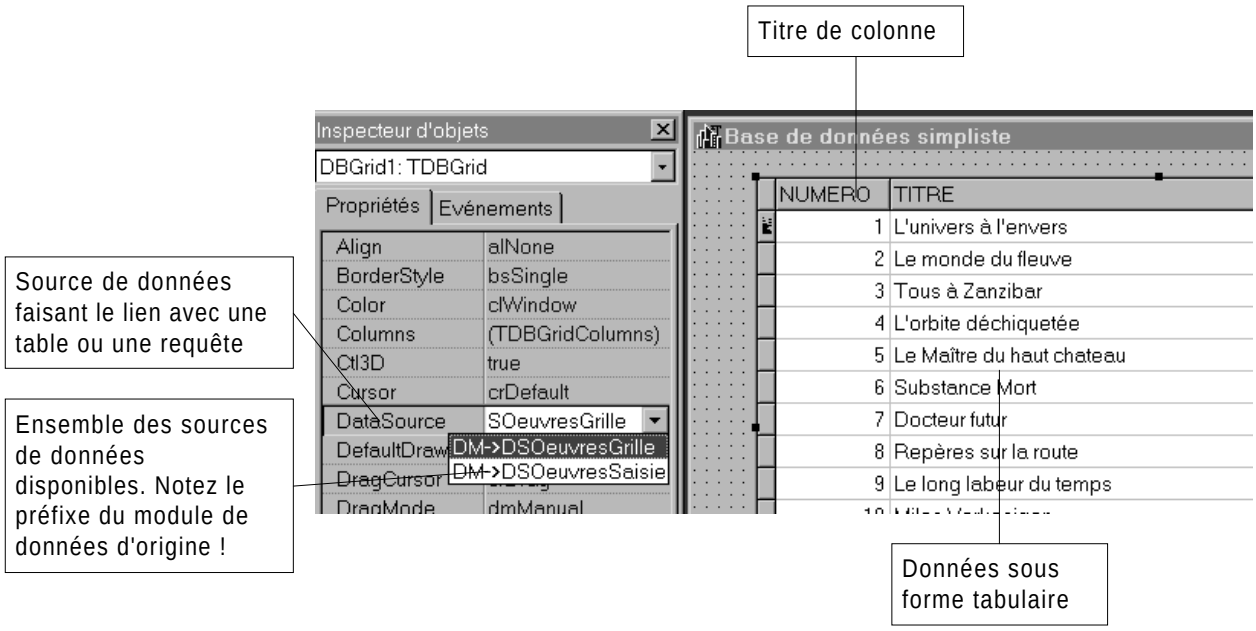


Figure 4.6 Présentation tabulaire des données dans un composant TDBGrid

La propriété la plus importante d'une **TDBGrid** est **Columns** qui est principalement constituée d'un tableau de **TColumn** dont chaque élément est représentatif d'une colonne du tableau.

A sa création, un composant TDBGrid est doté de colonnes dynamiques, c'est à dire de l'ensemble des champs de la relation avec les formats prédéfinis. Il est toutefois possible de créer des colonnes statiques permettant – entre autre – des présentations plus sophistiquées des données. Toutefois, il faut garder à l'esprit que

les colonnes dynamiques « bougent » avec l'ensemble de données. En particulier, si vous ajoutez de nouveaux champs (ou si vous en supprimez ☺), les colonnes dynamiques suivront cette progression alors qu'il vous faudra modifier les colonnes statiques.

Le meilleur moyen de créer des colonnes statiques consiste à double cliquer sur la grille pour faire apparaître la liste des colonnes statiques (initialement vide). Ensuite, on gagnera à activer « Tous les champs » pour ajouter à la listes des colonnes statiques l'ensemble de champs actuellement présents dans la Table. Les différents boutons

Les différents boutons de la fenêtre des colonnes statiques permettent de changer l'ordre des colonnes, de supprimer celles que l'on veut exclure de l'affichage, ou bien d'ajouter de nouvelles colonnes si l'on a touché à la structure de la table (ou de la requête) ou si l'on a précédemment supprimé une colonne par inadvertance (ou nervosité ☺).

Une fois la liste des colonnes statiques créée, il est possible de manipuler chaque colonne dans l'inspecteur d'objets comme le montre la figure suivante.

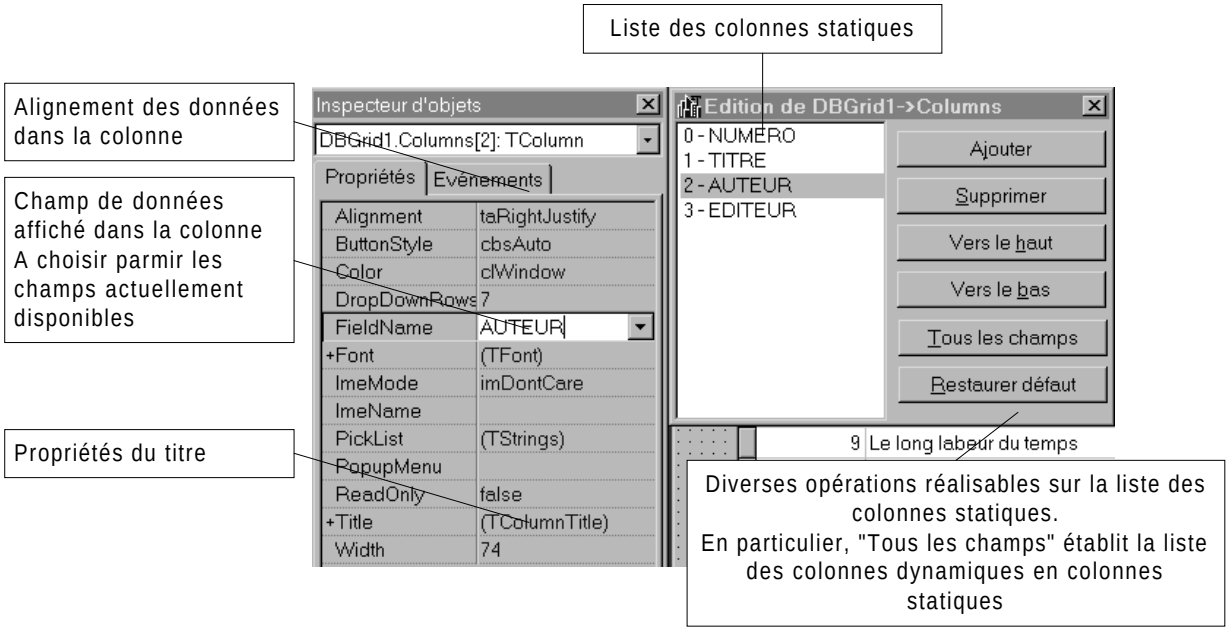


Figure 4.7 Manipulation des colonnes

La propriété **`Title`** est très complexe et comprend tout ce qui est nécessaire à la manipulation d'un titre, y compris une propriété **`Font`** permettant de spécifier un style de caractères différent de celui de la colonne. En particulier, il est possible d'aligner différemment les données de la colonne et leur titre ou de choisir une fonte différente.

Initialement, le champ **`Title`** reprend le nom du champ de données ce qui n'est guère agréable, il est tout à fait possible de le modifier !

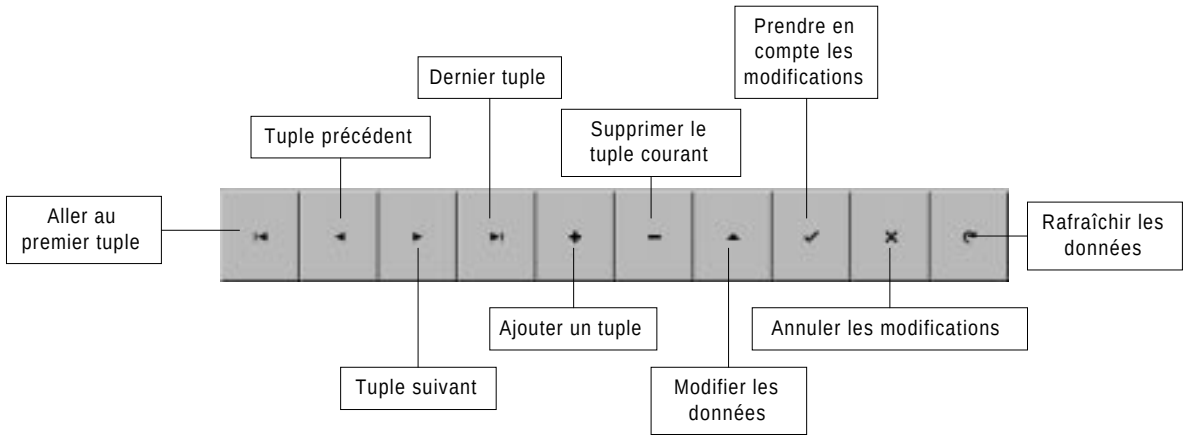
A titre d'exercice, changez le titre des colonnes de manière à ce qu'il reprenne le nom du champ mais en gras et en minuscules.

4.3.2 Les autres contrôles

Les autres contrôles permettent essentiellement de modifier ou d'ajouter de nouveaux éléments dans la base de données. Par exemple, les composants liste orientés bases de données autorisent l'utilisateur à changer la valeur d'un champ dans un tuple existant ou en cours de création.

A l'usage, ils s'avèrent relativement dangereux car ils effectuent des opérations d'insertion ou de prise en compte automatique des modifications particulièrement nocives à la santé. Aussi, je ne préconise l'utilisation que de deux composants de saisie orientés données : **`TDBLookupComboBox`** (que nous étudierons un peu plus loin) et **`TDBNavigator`**.

Les composants navigateur ou TDBNavigator permettent à un utilisateur de se déplacer dans un ensemble de données et d'y effectuer des modifications si ceci est possible. Sous sa forme la plus complète, ce composant se présente ainsi :



4.4 Manipulations élémentaires sur les bases de données

C++ Builder autorise en direct quelques opération élémentaires de l'algèbre relationnelle, à savoir :

Opération d'algèbre relationnelle	Vocabulaire et commande C++ Builder
Jonction	Ajout de champ « Référence » dans un ensemble de données
Sélection	Filtrage ou création de relations Maître / Détail
Projection	Création de colonnes statiques et suppression des champs non désirés dans les **`TDBGrid`** (déjà vu)

Tableau 4.1 Correspondance entre opérations d'algèbre relationnel et opérations C++ Builder

4.4.1 Réalisation de jonctions

Dans notre exemple, nous aimerions voir apparaître les noms de l'auteur et de l'éditeur en lieu et place de leurs numéros de référence dans la représentation tabulaire de la relation œuvres. Pour cela, nous allons ajouter des champs dits de référence dans la relation. Ceux-ci ne sont pas autre chose que des jonctions limitées.

L'accès aux champs d'un ensemble de données est faussement similaire à celui des colonnes d'un **TDBGrid**, aussi un avertissement solennel s'impose. Il faut tout de suite que vous fassiez la différence entre les champs présents dans un **TDataSet** et la liste des colonnes de présentation présentes dans un **TDBGrid**. Le premier représente la liste des champs présents dans l'ensemble de données, que soit des champs endogènes (présents dans le fichier pour une table ou dans la définition pour une requête) ou des champs exogènes ajoutés par jonction ou calcul. Le second ne définit que des options d'affichage de données. Le problème vient du fait que le vocabulaire est très similaire.

Une fois de plus, il nous faut distinguer entre champs dynamiques et statiques. Afin d'accéder à la liste des champs statiques présents dans un ensemble de données, il est nécessaire de double cliquer sur son icône ce qui finit d'accentuer la confusion possible avec la liste des colonnes d'un **TDBGrid**.

Le menu contextuel possède deux options fort intéressantes :

- « Ajouter champs » permet d'ajouter dans la liste des champs statiques un champ de donné présent dans la structure du **TDataSet**.

- « Nouveau champ » permet d'ajouter un nouveau champ par jonction ou calcul.

Précisons immédiatement que nous aurons le droit d'ajouter plusieurs types de champs.

- Les champs de données sont les champs endogènes de la table, cette option ne présente que peu d'intérêt car vous pouvez faire la même chose avec le menu « Ajouter champs »

- Les champs calculés permettent d'effectuer des opérations arithmétiques ou lexicographiques sur les différents champs. Par exemple, vous pourrez effectuer une addition entre deux champs numériques ou une concaténation de deux champs chaînes de caractères.

- Les champs de référence permettent d'effectuer une jonction c'est-à-dire une mise en correspondance des valeurs de deux champs dans deux tables différentes avec extraction d'un résultat.

Ajoutons un champ nommé NOM_AUTEUR et spécifiant le nom de l'auteur en à partir de son numéro. La jonction s'effectuer entre les champs AUTEUR de « œuvre » et NUMERO de « auteur » avec NOM de « auteur » pour résultat. Pour parler avec le langage des bases de données, NUMERO est la clef primaire de la relation « auteur » alors que AUTEUR est une clef *étrangère* (ou *externe*) de « œuvre ».

La boîte de dialogue obtenue après activation de « Nouveau champ » est la suivante :

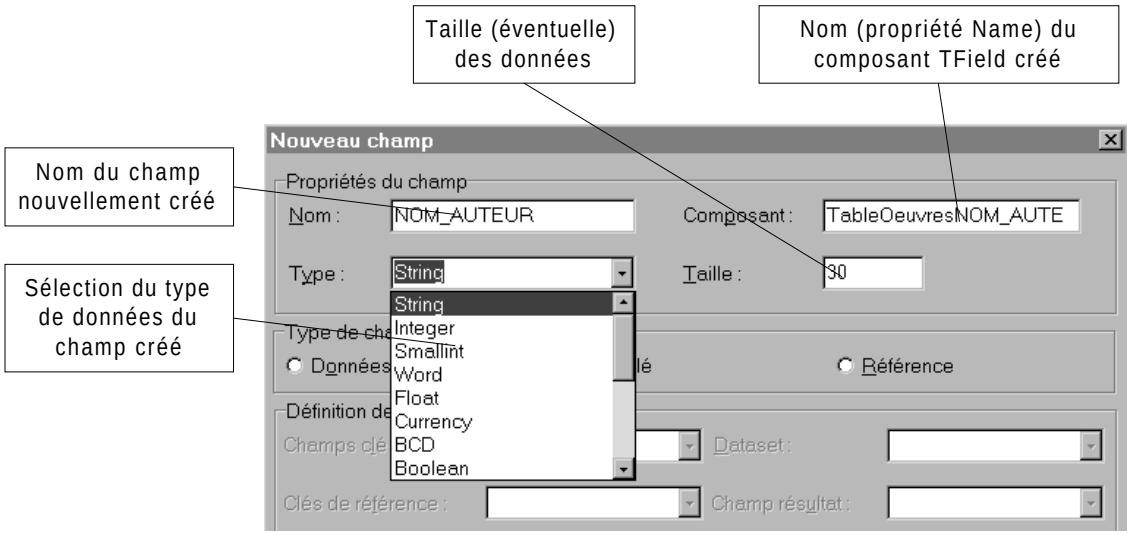


Figure 4.8 création d'un champ de données

Les premières spécifications à apporter sont :

- **Le nom du champ** que vous désirez ajouter. C++ Builder gère pour vous le nom du composant **`TField`** généré pour représenter votre nouveau champ.

- **Le type de données du champ** et éventuellement sa taille. Dans le cas d'un champ calculé tout type et autorisé. Dans le cas d'un champ de données ou dans le cas d'un champ référence en provenance d'un autre ensemble de données, il faut faire attention à n'utiliser que des types compatibles avec les données sources, par exemple, des données numériques entre elles.

Continuons notre jonction en choisissant de créer un champ référence. Le résultat de la manipulation est illustré par la figure suivante :

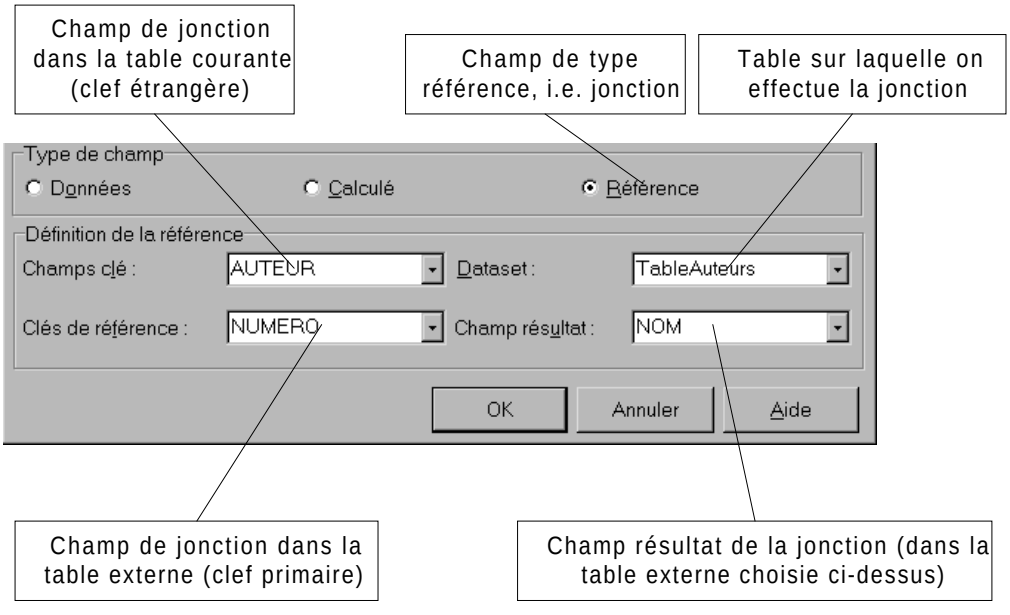


Figure 4.9 Réalisation d'une jonction : étape 2

- Choix du champ clef : C'est un champ existant dans la table actuelle et qui doit être une clef étrangère, dans notre cas AUTEUR. C'est l'un des deux champs sur lesquels la jonction va être opérée.

- Choix de l'ensemble de données : c'est l'ensemble de données sur lequel va être effectuée la jonction, ici « auteurs ».

- Clef de référence : c'est le deuxième champ de jonction. Il est préférable que ce soit la clef primaire ou, au minimum, un champ sur lequel est défini un index secondaire dans l'ensemble de données de jonction, ici NUMERO.

- Champ résultat : c'est le résultat de la jonction, à un champ de l'ensemble de données cible (projection), ici NOM.

Le champ nouvellement créé peut alors se traiter comme s'il avait toujours appartenu à la table (champ endogène) à quelques exceptions près, comme, par exemple, les opérations de filtrage.

Exercice (évident):

Réaliser la même opération pour le nom de l'éditeur.

4.4.2 Le filtrage

Le filtrage est une opération qui permet de restreindre les tuples d'une table à ceux qui respectent une certaine condition nommée Filtre. Pour activer un filtre, il faut spécifier dans la propriété **Filter** une chaîne de caractères spécifiant les critères de filtrage et positionner la propriété **Filtered** à **true**. Ces propriétés étant modifiables à tout instant, cette opération pourra s'effectuer très facilement et sous le contrôle d'éléments d'interface.

Seuls les champs endogènes d'un ensemble de données peuvent être filtrés.

4.4.3 Création de fiches Maître / Détail

Supposez désormais que vous souhaitiez accéder aux données d'un ensemble en fonction du tuple sélectionné dans une grille. Le meilleur exemple est sans doute celui des nomenclatures où vous disposez des trois tables.

- La table « produits finis » représente l'ensemble des produits complexes

- La table « matières premières » représente l'ensemble des composants élémentaires que vous assemblez

- La table « composition » associe sur une ligne, un produit complexe, un composant simple et le nombre de composants simples entrant dans la composition du produit complexe.

Vous pouvez alors décider d'afficher dans un premier TDBGrid l'ensemble des produits complexes, puis, dans un second, la liste des pièces le composant. Il s'agit

typiquement d'une liaison maître / esclave. La table « produits finis » (le maître) conditionnant l'accès aux données de « composition » (l'esclave).

Pour en revenir à nous bouquins de science fiction, considérons désormais que chaque œuvre puisse maintenant avoir plusieurs auteurs. Il n'est plus alors question d'utiliser un simple champ permettant de référence l'auteur : il nous faut utiliser une nouvelle relation que nous appellerons « **assoc** » pour Association Œuvre / Auteurs et qui associe sur une même ligne une référence d'œuvre à une référence d'auteur. Il suffira d'ajouter autant de lignes pour un même œuvre qu'elle compte d'auteurs.

Procédons à la mise en place de la relation Maître / Esclave permettant d'afficher, pour une œuvre sélectionnée dans un premier **TDBGrid** l'ensemble de ses auteurs. Typiquement, le maître sera la table « œuvres », alors que l'esclave sera « assoc ».

Une fois les données créées, voici la marche à suivre :

- Préparation de la visualisation de la relation maître / esclave
  - Ajouter un nouveau composant **TTable** et une source de données à la relation « assoc ».
  - Ajouter le champ NOM_AUTEUR à la table des associations par jointure avec « auteur » sur le numéro de l'auteur.
  - Associer une grille à la table des associations et ne présenter que le nom de l'auteur.
- Mise en place de la relation Maître / Détail. Celle-ci se fait essentiellement sur les propriétés de la table esclave
  - Sélectionner la propriété **MasterSource** de la **TTable** assoc et lui donner pour valeur la source de données associée à la **TTable oeuvres**. Ceci indique clairement que nous nous baserons sur l'élément courant de cette source de données comme maître.
  - Utiliser l'éditeur spécial pour choisir les **MasterFields**, opération qui est similaire à une jointure entre un champ de la table maître et un champ de la table esclave
    - Choisir un index de « assoc » pour obtenir le champ joint côté esclave, ici TITRE qui dénote un numéro d'œuvre.

      Notez au passage que les relations maître / esclave ne peuvent avoir lieu sans indexation correcte de la table esclave.
    - Choisir le champ de jointure côté « œuvres » dans la case de droite, ici NUMERO qui référence le numéro de l'œuvre (clef primaire)
    - Cliquer sur « Ajouter », le tour est joué !

Les figures suivantes illustrent en deux étapes la création de la liaison maître / esclave aussi nommée maître / détails.

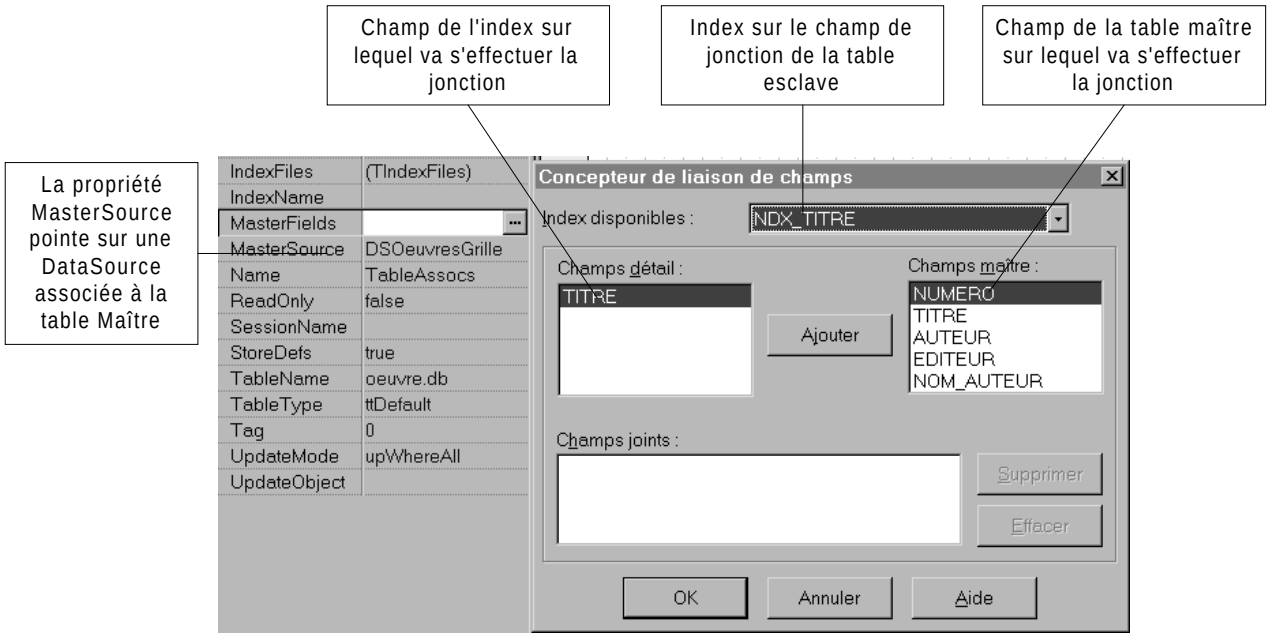


Figure 4.10 Création d'une liaison Maître / Détail : Mise en place

Appuyer sur le bouton « Ajouter » ajoute le couple sélectionné à la liste des champs de jointure qui peut compter jusqu'à 16 couples !

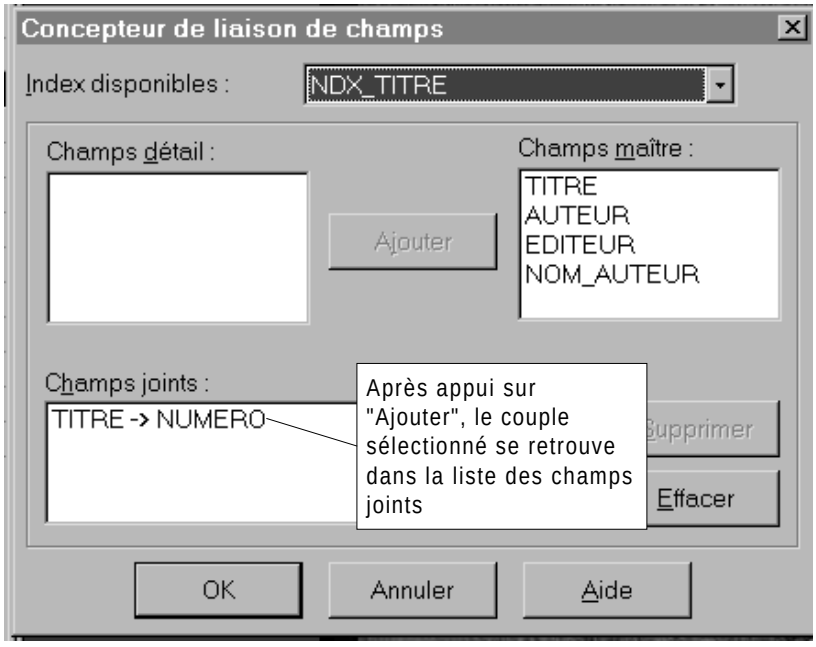


Figure 4.11 Fin de création d'une liaison Maître / Détail

4.4.4 Ajout d'un tuple dans une ou plusieurs table

Pour ajouter un tuple dans une table, il suffit de passer la table dans le mode *insertion*, ce qui a pour effet de créer un nouveau tuple vierge. Dans un deuxième temps, on modifie les valeurs des champs par une instruction du style :

```
IdentificateurTable.FieldByName("nom du champ") = Valeur ;
```

Cette opération peut être réalisée directement en affectant un contrôle de données à un champ. Par exemple, une boîte d'édition de données **TDBEdit** est connectée à un champ endogène d'une table via sa source de données. Cependant cette opération est assez dangereuse et il vaut mieux utiliser des champs simples et confier à un bouton genre « Valider » le soin d'incorporer les données dans la base.

Deux commandes principales permettent de passer en mode insertion : **Append** et **Insert**. Il est toujours préférable d'utiliser la première qui est moins traumatisante pour la base de données. En effet, il n'est guère intéressant de rajouter des tuples au milieu d'une base lorsque l'on peut les obtenir dans l'ordre que l'on veut en utilisant un index approprié, n'est ce pas ?

Certains contrôles permettent de sélectionner des champs en effectuant une jonction, c'est le cas, par exemple des deux composants **TDBLookupListBox** et **TDBLookupComboBox**.

Replaçons nous dans le cas où chaque œuvre peut n'avoir qu'un seul auteur, nous allons effectuer l'opération de saisie :

- Il est possible de saisir librement le titre de l'œuvre et le nom de l'auteur, ce qui se fera par l'intermédiaire de simples composants **TEdit**.
  - Si l'auteur est déjà présent dans la base « auteur », alors son numéro est extrait de cette dernière et incorporé dans « œuvre »
  - Sinon, un nouvel auteur est rajouté dans « auteur » avant que son numéro ne soit ajouté à « œuvre »
- Il est interdit de rajouter un nouvel éditeur : celui-ci devra être choisi parmi ceux déjà présents dans la base « editeur ». Nous utiliserons pour cela une **TDBLookupComboBox**.

Nous allons construire une fiche sur le modèle suivant :

La partie de gauche est dédiée au titre de l'œuvre, elle est composée d'un **TDBEdit** permettant de saisir le titre, d'un bouton nommé « ajouter » validant cette action et d'un bouton nommé « annuler ».

Une fois le titre validé, il sera possible de rajouter autant d'auteurs que nécessaire en les sélectionnant un par un dans une liste (**TDBLookupListBox**) et en appuyant sur un bouton nommé « ajouter ». Il ne sera pas possible d'annuler avant d'avoir ajouté au moins un auteur.

Le résultat de l'interface est présenté sur la figure suivante :

Le composant **TDBEdit** est fort simple, il suffit de le connecter sur le bon champ via une source de données (en l'occurrence, la source de données est DSOeuvres et le

champ TITRE). En revanche la **TDBLookupListBox** est plus difficile à utiliser car elle utilise deux ensembles de données différents mais devant appartenir à la même base de données, la figure suivant résume ses principales propriétés :

- Les propriétés **DataSource** et **DataField** sont respectivement reliées à la source de données et au champ dans lequel on souhaite écrire une valeur.
- Les propriétés **ListSource** et **ListField** sont elles liées à la source de données et au champ à lister.
- Quand à la propriété **KeyField**, c'est un champ en provenance de **ListSource** et qui correspond à la seconde partie de la jonction. Sa valeur sera recopiée dans celle de **DataField**. Ainsi, c'est **ListField** qui est présenté et **KeyField** la valeur recopiée dans le champ en cours de modification.

Les méthodes **Post** et **Cancel** de **TTable** permettent respectivement d'appliquer des changements ou de les annuler.

Exercice : (✶✶)

Rajouter la possibilité d'avoir plusieurs auteurs.

5. Utilisation du Module de Bases de Données

5.1 Le module de bases de données

C'est un utilitaire très pratique fourni en standard avec les outils de développement de chez Inprise et qui remonte à la lointaine époque de Borland.

Cet utilitaire permet, d'éditer les données présentes dans des tables déjà existantes à l'aide d'une interface minimaliste. Ce n'est toutefois pas là qu'il exprime tout son potentiel.

En effet, il permet de créer des tables avec leurs index éventuels et même, dans certains cas, de générer des contraintes d'intégrité référentielle. Ceci permet de retrouver dans C++ Builder certaines des fonctionnalités qui lui manquaient par rapport à Access. Il faut toutefois insister sur le fait que le Module de Bases de Données n'est qu'un utilitaire qui ne saurait en aucun cas se comparer à un système complet de gestion de bases de données tel qu'Access.

Nous étudierons par la suite deux des fonctionnalités les plus utiles du MDD : la création de tables et la création de requêtes SQL.

5.2 Présentation de l'interface

C'est assurément la fonctionnalité la plus utilisée du MDD. En effet, elle permet de créer les tables sur lesquelles on veut travailler, fonctionnalité que C++ Builder n'est pas capable de réaliser sans programmation.

La création est lancée par le menu Fichier → Table → Nouveau

La première étape consiste à sélectionner le type de table dans la boîte combo qui apparaît à l'écran (voir la figure suivante) Si la portabilité des données n'est pas votre préoccupation première, je vous conseille le format Paradox 7 pour lequel BDE (Borland Database Engine, le moteur de bases de données de C++ Builder) est optimisé.

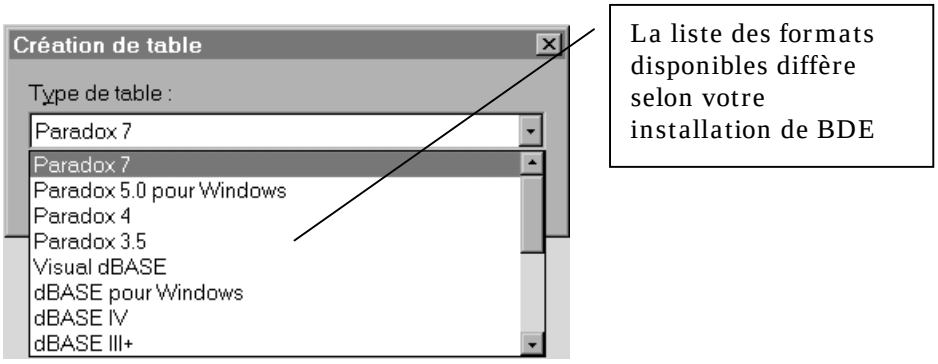


Figure 5.1 Liste des formats disponibles en création de table

Une fois le type de table sélectionné, le MDD vous affiche sa fenêtre principale telle qu'elle est présentée par la figure suivante :

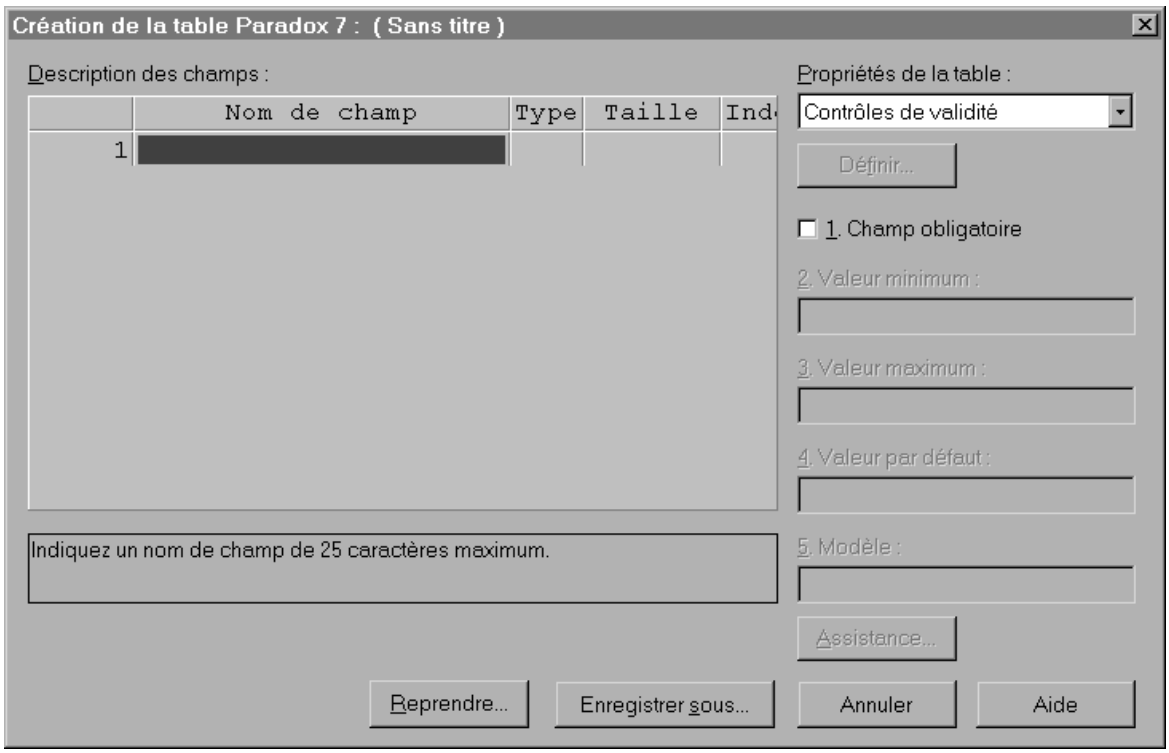


Figure 5.2 Fenêtre principale du MDD

Cette fenêtre se divise en deux grandes parties. A gauche, vous avez la liste des champs de la table, dans leur ordre de stockage, accompagnés de leur type, de la taille si elle s'applique ainsi que d'une colonne nommée Index et occupée par une étoile si le champ fait partie de la clef primaire.

La partie droite est plus versatile et dépend de l'option choisie. La figure suivante regroupe les différentes rubriques accessibles auxquelles il faut rajouter « Langage de Table ».

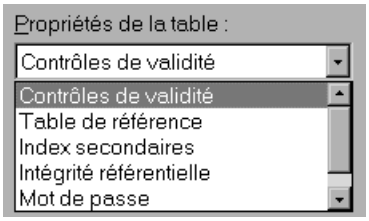


Figure 5.3 liste des options disponibles

Les rubriques les plus utilisées sont :

Contrôles de validité : permet de spécifier des contraintes sur l'attribut, en particulier, imposer que sa valeur soit instanciée. Il est également possible de fournir des valeurs extrêmes pour les attributs.

Index secondaires : permet d'ajouter des index secondaires sur la table. Cette fonctionnalité sera traitée plus en détail dans une section ultérieure.

Intégrité référentielle : permet de fixer des contraintes d'intégrité référentielle (on s'en doutait un peu ☺) entre eux tables. Pour résumer, la

valeur d'un champ dans une table B doit correspondre à une valeur existante de la clef primaire d'une autre table A.

5.3 Ajout de champs

L'ajout de champs se fait tout simplement en renseignant les rubriques de la partie gauche de la fenêtre, c'est à dire, dans l'ordre : le nom, le type, éventuellement la taille pour finir par la présence dans la clef primaire.

5.3.1 Le nom

La première donnée à saisir concerne le nom du champ. Les conventions de nommage dépendent du type de table que vous avez choisi. En règle générale, et même si certains systèmes le permettent, il vaut mieux ne pas utiliser d'espaces dans le nom d'un champ.

5.3.2 Le type

La figure suivante montre les différents types de données disponibles pour les tables de type Paradox 7. Cette liste s'obtient en tapant « espace » dans la case réservée au type des données.

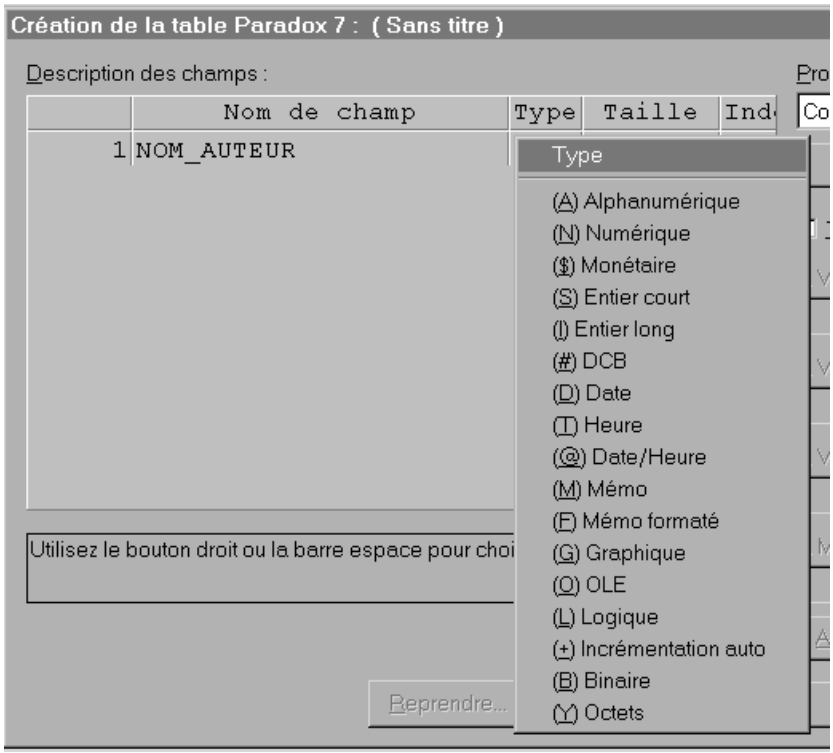


Figure 5.4 Types de données disponibles

La plupart des types de données présents ne pose aucun problème. Je voudrai toutefois revenir sur les suivants :

Incrémentation auto : la valeur de ce champ est calculée automatiquement à partir de la dernière valeur présente dans la table. Aussi, vous n'avez pas la possibilité de la fixer vous même. Ce type est parfait pour créer des clefs

primaires numériques si la structure de votre table n'admet pas de clef primaire naturelle.

Mémo / Mémo Formaté / Binaire / Graphique ces différents types sont destinés à stocker des quantités importantes de données. Contrairement aux autres champs, ils ne sont pas stockés à leur place dans la table mais tous ensemble, soit dans un fichier séparé, soit tout à la fin du fichier. Leur accès est ainsi soumis à des règles précises.

5.3.3 La taille

Certaines données, telle que les chaînes de caractères et les données numériques en virgule fixe nécessitent une taille. Elle se présente soit sous la forme d'un entier, soit d'une paire d'entiers (taille et précision pour les réels en virgule fixe)

5.3.4 La présence dans la clef primaire

Si vous souhaitez qu'un champ soit présent dans la clef primaire, il faut saisir une étoile dans la dernière colonne de la fenêtre comme présenté par la figure suivante. Attention aux bugs du MDD, n'essayez surtout pas de saisir un autre caractère, cela peut faire planter sauvagement Windows (GPF ☺). Bien entendu, il est possible de faire figurer plusieurs champs dans la clef primaire.

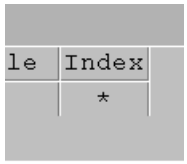


Figure 5.5 la case Index

5.4 Définir des index secondaires

Les index secondaires permettent d'accéder très rapidement aux données en fonction d'un certain critère sur les données. La figure suivante montre l'aspect de la fenêtre du MDD lorsque l'on sélectionne Index secondaires dans les options.

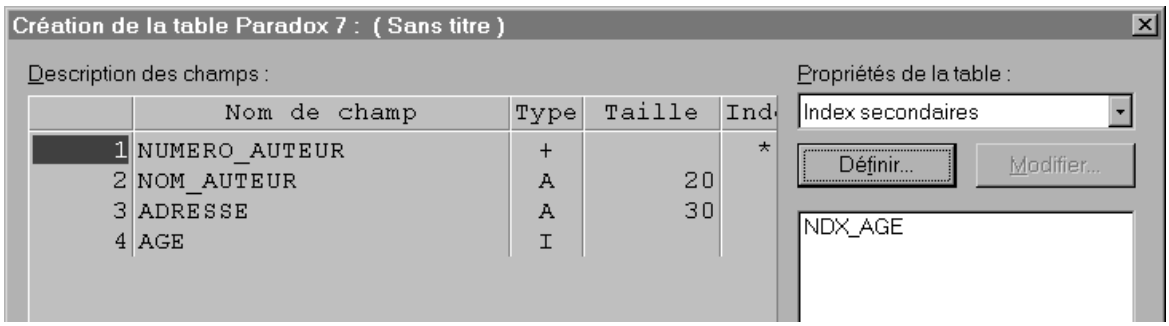


Figure 5.6 la liste des index secondaires

La liste sur la droite indique quels index secondaires sont déjà disponibles. Il est possible d'ajouter des index, de modifier ou de supprimer les index déjà existants.

La fenêtre suivante est associée aux opérations d'ajout ou de modification d'index secondaire.

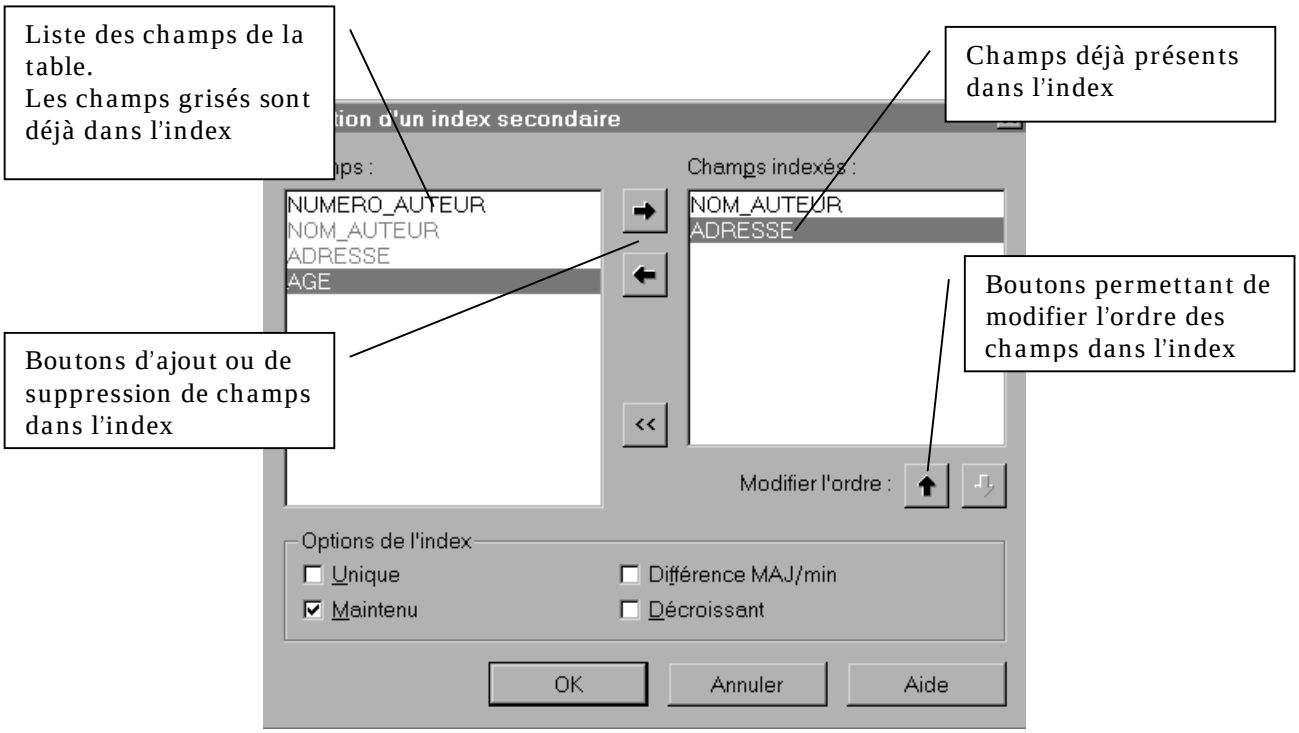


Figure 5.7 Définition des index secondaires

Les champs apparaissant en grisé dans la liste de gauche sont déjà inclus dans l'index et ne peuvent donc plus y être ajoutés.

Lorsque l'on spécifie plusieurs champs dans un index, celui-ci est multicritères hiérarchique. Les tuples sont ordonnés d'abord en fonction du premier champ, puis, pour chaque valeur du premier champ, en fonction du second, puis pour chaque valeur identique du couple (premier champ, second champ) en fonction du troisième et ainsi de suite.

Les options de cette fenêtre sont très simples à comprendre :

Unique : spécifie que c'est un index sans collision : pour une même valeur d'index, il ne peut y avoir qu'un seul tuple

Maintenu : indique que l'index est mis à jour à chaque opération sur la table. Ceci peut s'avérer coûteux si de nombreuses opérations sont effectuées mais garantit les meilleurs temps d'accès aux données.

Décroissant : par défaut, les tuples sont rangés par ordre croissant sur l'index, à moins que cette option ne soit cochée !

Différence MAJ/min : introduit une différenciation des mots selon la casse. Rappelons que les Majuscules sont alors prioritaires par rapport aux minuscules (dans l'ordre croissant)

6. Utilisation des contrôles ActiveX

Ce chapitre a pour objet l'inclusion et l'utilisation de contrôles ActiveX à l'intérieur d'une application C++ Builder. Comme vous allez le voir, hormis quelques phases préliminaires et la redistribution des applications, leur utilisation est absolument identique à celle des contrôles VCL standard.

6.1 Mise en place

La première opération consiste à vérifier l'installation du contrôle dans la base de registres de Windows. Ceci peut se faire grâce à une boîte de dialogue spécialisée de C++ Builder activée par le menu Composant → Importer un contrôle ActiveX.

6.1.1 Edition de la liste des contrôles disponibles

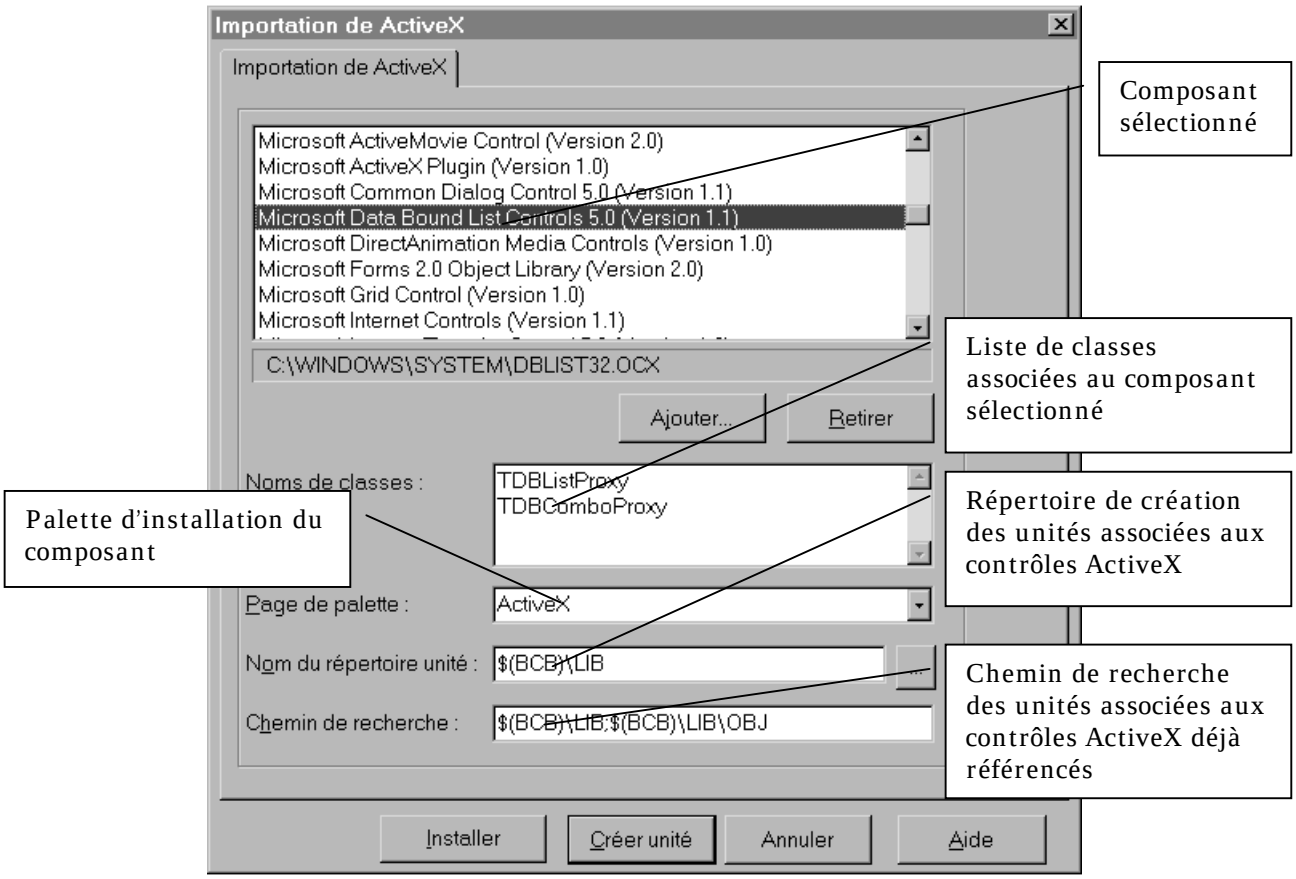


Figure 6.1 La boîte de dialogue des composants Active X

La première partie de la fenêtre est occupée par la liste des composants ActiveX installés sur votre ordinateur. Lorsque vous sélectionnez un composant, deux cas peuvent se produire :

- Une unité est déjà associée au composant, auquel cas, un ou plusieurs noms de classes lui sont associées et apparaissent au centre de la boîte de dialogue.

Dans ce cas, le composant est près à être utilisé et vous pouvez vous rendre sereinement à la section 2 !

- Aucune classe n'apparaît il vous faut alors associer un nom de classe au composant et créer une unité référençant ses capacités. Cette activité est commentée en détails dans la section « Création d'une unité ».

  Néanmoins, il se peut que votre composant soit déjà prêt à l'emploi mais que C++ Builder n'ait pas été en mesure de trouver vos unités si le chemin de recherche (spécifié par la boîte de saisie en bas de la fenêtre de dialogue) est erroné. Il est donc nécessaire de vérifier ce dernier si vous pensez qu'un autre utilisateur de votre système a déjà effectué le travail de création d'une unité.

Si le composant que vous désirez utiliser n'apparaît pas dans la liste, cela signifie tout simplement qu'il n'est pas référencé dans la base de registre ; opération que nous allons maintenant expliciter.

6.1.2 Recensement dans la base de registres d'un nouveau composant

L'installation d'un nouveau composant se fait en appuyant sur le bouton Ajouter de la boîte de dialogue précédente. Une nouvelle boîte de dialogue vous invite alors à sélectionner le fichier contenant votre contrôle, ce dernier a pour extension **.ocx**.

A titre d'exemple, nous allons installer le contrôle standard de calendrier de Microsoft associé au fichier c:\activex\mscal\mscal.ocx. Une fois le fichier sélectionné et valider, la boîte de dialogue précédente présente l'aspect suivant :

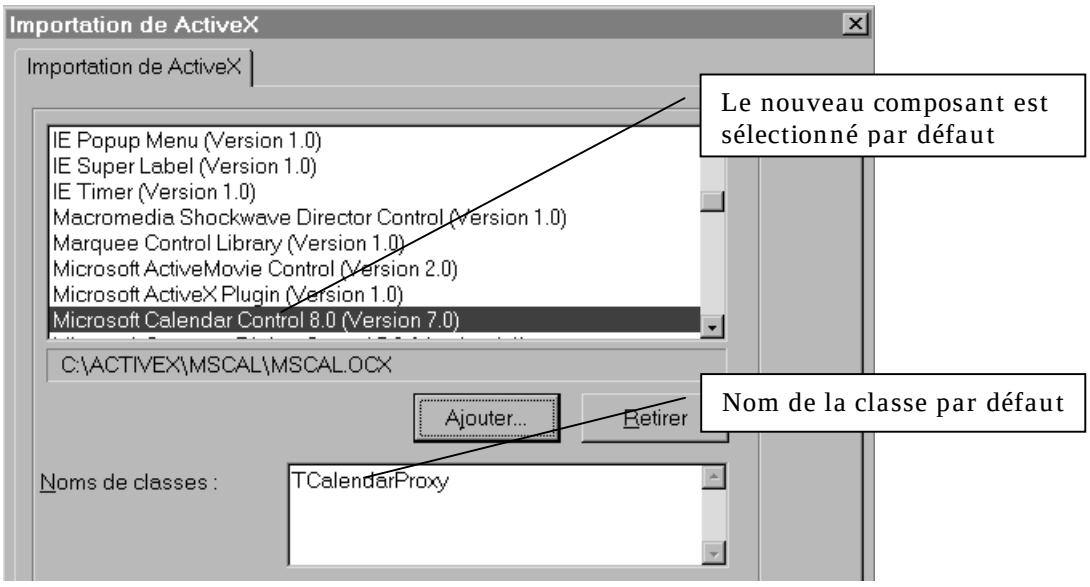


Figure 6.2 Fenêtre des composants ActiveX après enregistrement d'un composant

Vous noterez que le nouveau composant est dorénavant présent dans la liste. Le nom est celui d'enregistrement dans le fichier .ocx accompagné du numéro de version

simplifié. Le numéro de version complet peut être obtenu grâce à l'utilitaire Ole2View disponible sur le site ouaib de qui vous savez.

C++ Builder propose toujours un nom de classe finissant par **`Proxy`**. En effet, en terminologie OLE, on appelle classe **`Proxy`**, toute classe représentant un objet Automate OLE ou ActiveX et masquant l'utilisation des diverses interfaces. Utiliser le nom par défaut présentes les avantages de la clarté et de l'auto-documentation ; toutefois, rien ne vous empêche de le changer.

La prochaine étape consiste à ajouter à votre système une unité (i.e. une combinaison d'un fichier `.h` et du fichier .cpp associé) qui permettront d'utiliser facilement votre composant dans une application.

6.1.3 Création d'une unité

Cette étape permet de créer deux fichier : un fichier d'entête et un fichier d'implémentation à même de faciliter l'utilisation d'un composant ActiveX dans une application. En outre, nous allons ajouter le composant dans une palette afin de pouvoir l'utiliser facilement. Par défaut, C++ Builder vous propose d'installer le composant dans la palette intitulée ActiveX, ce qui paraît naturel. Toutefois, il vous est possible de spécifier une autre palette dans la boîte de saisie « Page de palette ».

La localisation des fichiers unités est définie par la boîte de saisie intitulée « Nom du répertoire unité ». Celle-ci pointe naturellement sur le répertoire où C++ Builder stocke ces fichiers de librairie. Personnellement, je recommande de créer un autre répertoire (par exemple, `c:\activex\imports`) où vous logerez toutes vos nouvelles unités ActiveX. Afin que C++ Builder s'y retrouve, n'oubliez pas d'ajouter ce répertoire au chemin de recherche des unités situé dans la boîte de saisie du dessous.

Le processus de création de l'unité est lancé par l'activation du bouton « Installer » de la boîte de dialogue. Une fois le bouton enfoncé, la boîte de dialogue ActiveX est fermée pour être remplacée par la boîte de modifications des paquets.

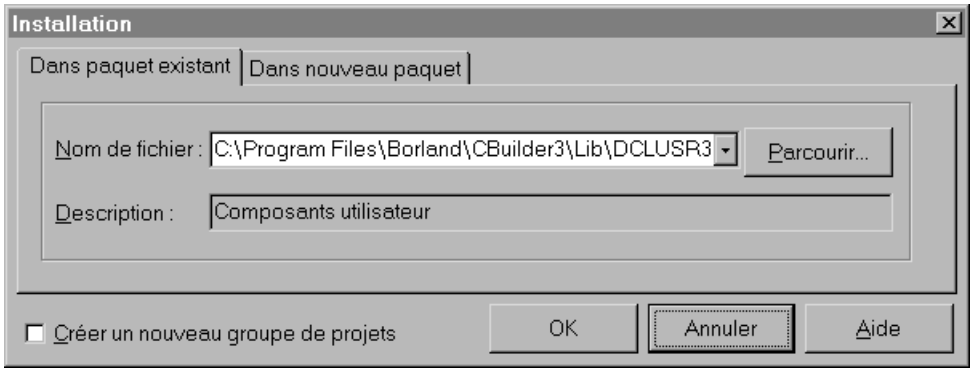


Figure 6.3 Installation du composant dans un "paquet"

La notion de « paquet » est quelque peu compliquée pour être expliquée ici. Pour l'heure il vous suffit de valider systématiquement toutes les options qui vous sont présentées.

La nouvelle unité sera alors compilée et enregistrée dans un « paquet », la dernière boîte de message vous indiquant que l'opération a été réalisée avec succès – du moins, je l'espère – et que votre composant a été recensée avec le nom de classe que vous aviez spécifié.

Lorsque toutes les boîtes de dialogue successives ont été fermées, deux nouveaux fichiers apparaissent dans la fenêtre d'édition. Leur nom est composé pour partie du nom du composant ActiveX suivi d'un tiret bas et du mot TLB. Ce dernier signifie Type Library et indique ainsi que les fichiers créés contiennent la librairie de type du composant ActiveX.

Rappelons brièvement que la librairie de type contient la liste des interfaces associées à un composant, et, pour chaque interface :

- L'ensemble des méthodes chacune accompagnée de la liste de ses paramètres avec leurs types
- L'ensemble des propriétés avec leurs types

La consultation du fichier .cpp n'est habituellement d'aucune utilité. En revanche, l'utilisateur curieux peut se lancer – non sans s'être préalablement muni d'aspirine et d'Alka Seltzer™ en quantité suffisante – dans l'étude du `.h` qui ne manquera pas de lui montrer l'étendue des interfaces de son composant.

En outre, vous pourrez noter la présence d'une nouvelle icône dans la palette de composants que vous avez choisie pour recenser votre composant.

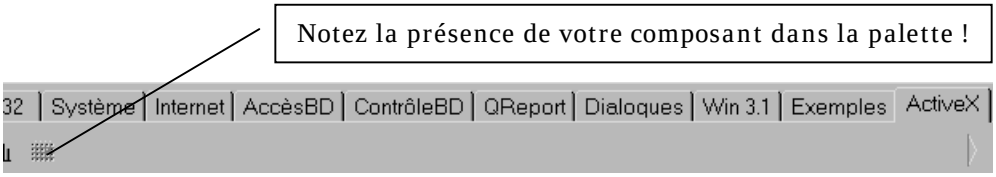


Figure 6.4 mise à jour de la palette

6.2 Utilisation d'un contrôle ActiveX

Une fois l'opération préliminaire réalisée, vous êtes pour ainsi dire sortie d'affaire. En effet, le contrôle ActiveX s'utilise comme n'importe quel composant de la VCL. Il est muni d'une représentation visuelle, et ces propriétés s'affichent tout à fait normalement dans l'inspecteur d'objet.

Dans la plupart des cas, vous pourrez lui associer des événements utilisateurs. Les figures suivantes montrent respectivement l'aspect du composant Calendrier posé sur une fiche et les événements qui lui sont associés. Les propriétés les plus intéressantes ont pour nom **`Day`**, **`Month`**, **`Year`** et **`Value`**. Je vous laisse les examiner vous même.

A partir de ces informations, je vous encourage à créer un petit programme mettant en scène ce composant !

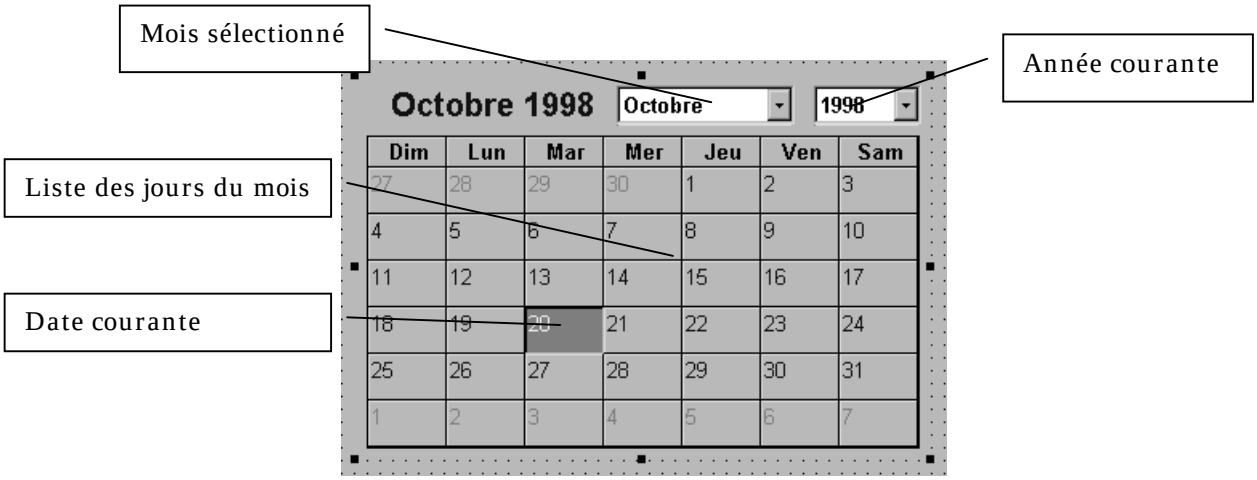


Figure 6.5 le composant Calendrier placé sur une fiche

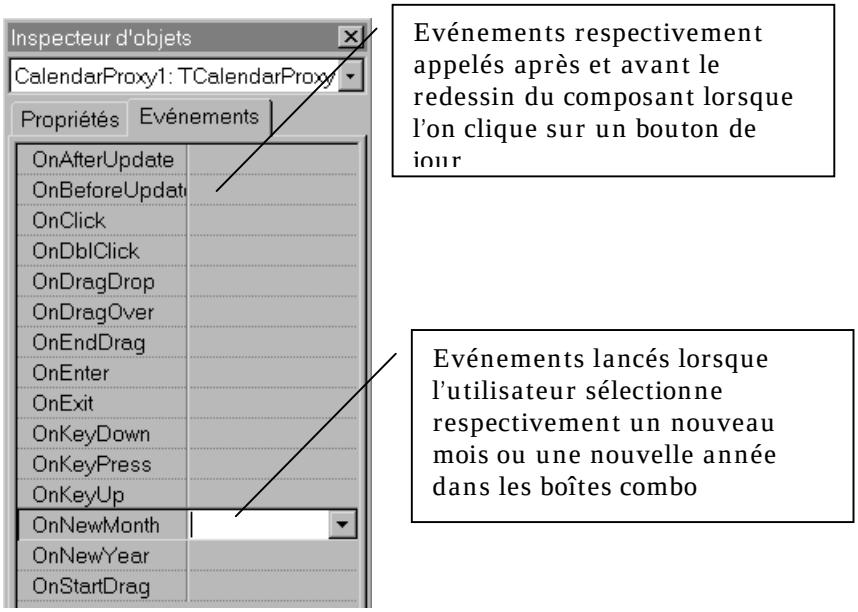


Figure 6.6 Evénements associés au contrôle calendrier

6.3 Déploiement d'un programme utilisant un contrôle ActiveX

L'utilisation d'un programme contenant un composant ActiveX nécessite l'installation de ce dernier. Aussi, il ne faut surtout pas oublier d'embarquer ledit composant lorsque vous déployez votre programme.

D'ordinaire, les fichiers associées à un composant sont les suivants :

- Un fichier `.ocx` qui contient le code du composant
- Un fichier `.hlp` et un fichier `.cnt`, supports de l'aide relative au composant
- Un fichier `.dep` de dépendances sur d'autres composants ActiveX

7. L'écriture de nouveaux composants

Nous abordons là l'utilisation avancée de C++ Builder. Tout d'abord, nous allons voir comment ajouter des composants à l'environnement et les adjoindre à une palette. Le chapitre suivant indiquera comment les transformer en composants ActiveX.

7.1 Généralités

Les composants sont des instances de classes dérivant plus ou moins directement de **TComponent**. Si leur forme la plus habituelle est celle des composants que l'on dépose d'une palette vers une fiche, ils englobent plus généralement la notion de brique logicielle réutilisable. Bien que non déclarée virtuelle pure[2] la classe **TComponent** n'est pas destinée à être instanciée. Compulsons sa documentation ;nous y apprenons que la plupart des méthodes sont protégés, c'est à dire inaccessibles à l'utilisateur, c'est une technique courante en Pascal Orienté Objet : définir le cadre de travail à l'aide de méthodes virtuelles protégées, lesquelles seront déclarées publiques dans les classes dérivées.

Autre aspect particulièrement intéressant : la présence des méthodes **AddRef**, **Release** et **QueryInterface** ce qui dénote l'implémentation de l'interface OLE **IUnknown**. Ainsi, lorsque l'on transformera un composant VCL en composant ActiveX, la gestion d'**IUnknown** sera directement prise en compte au niveau de **TComponent**. De la même manière, **TComponent** implémente l'interface principale d'automation **IDispatch** (méthodes **GetIDsOfNames**, **GetTypeInfo**, **GetTypeInfoCount** et **Invoke**). Nous aurons l'occasion de revenir plus en détails sur ces mécanismes spécifiques à OLE dans un prochain chapitre.

Pour finir, notons que les composants de Delphi et de C++ Builder sont compatibles : autrement dit, il est tout à fait possible d'utiliser dans C++ Builder un composant créé avec Delphi et réciproquement.

7.2 Création d'un nouveau composant

La création d'une nouvelle classe de composants se décompose globalement en les étapes suivantes :

1) Dérivation d'une classe existante

2) Implémentation de la classe

   a) Redéfinition des méthodes

[2] En effet, la classe **TComponent**, à l'instar de toutes les autres classes de la VCL est implémentée en langage Pascal Objet, lequel s'accommode assez mal de la notion de classe virtuelle pure.

b) Modification de règles de visibilité

c) Ajout de propriétés et d'événements

3) Test de la classe

4) Eventuellement, ajout d'un fichier d'aide pour les utilisateurs

5) Enregistrement dans le registre

6) Ajout dans une palette

Le processus de création d'un nouveau composant étant, par définition, très dépendant du concepteur, très peu d'étapes sont automatisables et réalisables par les fameux experts d'Inprise.

Toutefois, il est possible de gagner un tout petit peu de temps en invoquant *l'expert composant*.

Nous allons étayer notre propos par la réalisation de trois nouveaux composants. Le premier, (volontairement très simple) consiste à redéfinir le comportement de la classe **`TListBox`**, le deuxième à créer un composant d'actualité permettant de transformer les Euros en Francs et réciproquement. Le troisième composant sera de type graphique, spécialisé dans l'affichage de graphes. Nous montrerons également comment encapsuler une fiche d'intérêt général dans un composant afin de pouvoir la réutiliser plus facilement.

7.2.1 L'expert composant

Le début du processus consiste à demander la création d'un nouveau composant, via le menu Fichier → Nouveau puis à choisir Composant dans la fenêtre. La boîte de dialogue de l'expert composant est illustrée par la Figure 7.1

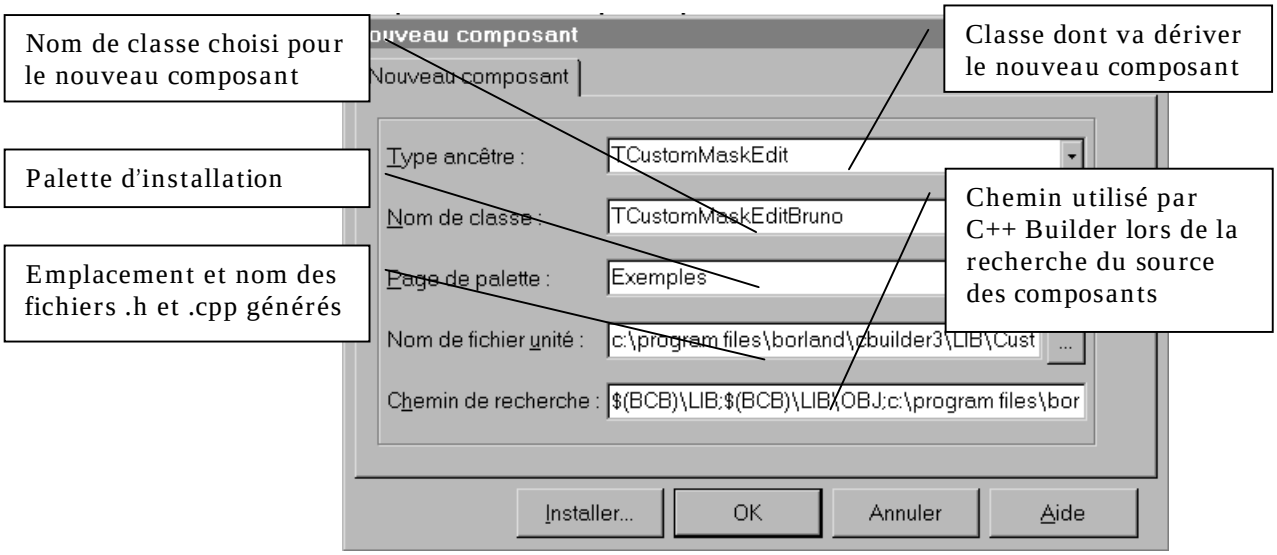


Figure 7.1 Expert composant

L'utilisation de cette fenêtre d'assistant est particulièrement limpide. Seul le bouton installer réclame notre attention et les options parlent d'elles mêmes.

Type Ancêtre : spécifie de quelle classe dérive votre composant, c'est le propos de la prochaine section

Nom de classe : saisissez le nom que vous souhaitez donner à votre classe de composant.

Page de palette : nom de la palette dans laquelle vous souhaitez inclure votre composant. Il est possible de créer de nouvelles palettes avec d'autres fonctionnalités de C++ Builder. Pour des raisons évidentes (embrouille des utilisateurs, manque de cohérence des composants présentés, etc.) je déconseille de rajouter un composant personnalisé dans une palette standard de C++ Builder

Nom de fichier unité : typiquement le nom des fichiers .h et .cpp générés reprend celui de la classe ce qui me paraît être une bonne idée. Plus intéressant, il vous est possible de choisir le répertoire dans lequel vous souhaitez installer ces fichiers. Par défaut, il s'agit d'un des répertoires appartenant à l'arborescence interne de C++ Builder. Je vous conseille de les positionner ailleurs afin d'éviter leur destruction lors d'une désinstallation *manu militari* de C++ Builder. Auquel cas, il convient de vérifier que votre nouveau répertoire est bien inclus dans le **chemin de recherche** spécifié dans la boîte de saisie suivante.

Finalement, seul le bouton Installer pose problème, son invocation amène la fenêtre suivante :

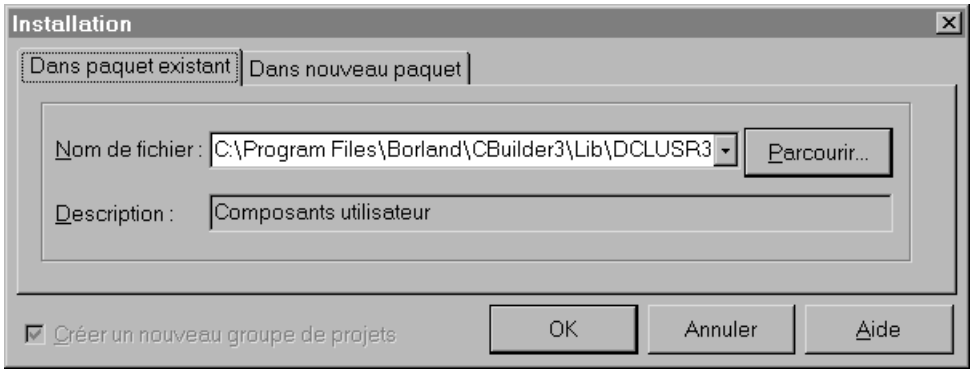


Figure 7.2 Spécification du paquet d'installation d'un nouveau composant

Son rôle est de spécifier le paquet dans lequel sera installé le composant. Brièvement, il suffit de savoir que les paquets correspondent à des DLL dans lesquels sont rassemblés les composants. Par défaut, tout nouveau composant est rajouté dans un paquet nommé DCLUSR3, c'est à dire « paquet utilisateur généré par la version 3 de C++ Builder » (ou, en Anglais *Delphi Cluster User Version 3*), lequel est situé dans les répertoires de l'arborescence de C++ Builder. Il vaut mieux utiliser un paquet situé dans un répertoire propre à l'utilisateur afin d'éviter toute désinstallation intempestive. De même, si vous désirez exporter votre composant, il sera probablement nécessaire de créer un nouveau paquet (vous disposez pour cela d'un onglet spécialisé) recueillant uniquement les composants destinés à votre client.

Nous verrons dans une prochaine phase que les paquets se manipulent très facilement à la manière des projets. En outre, si vous êtes dans le projet

correspondant à un paquet lorsque vous faîtes « nouveau composant » ce dernier est automatiquement placé dans le paquet associé au projet.

7.2.2 De quel composant dériver ?

La première question à se poser est la suivante : *« De quelle classe doit dériver mon composant ? »*

En effet, une grande partie du travail de conception dépend de cette phase initiale. Dériver d'une classe trop générale va vous contraindre à écrire énormément de code et à réinventer la roue. En revanche, dériver d'une classe trop spécialisée risque d'être coûteux en ressources Windows et à prévoir des comportements pour des propriétés ou des évènements sans intérêt pour vous. Le schéma de la Figure 7.3 permet de guider votre choix :

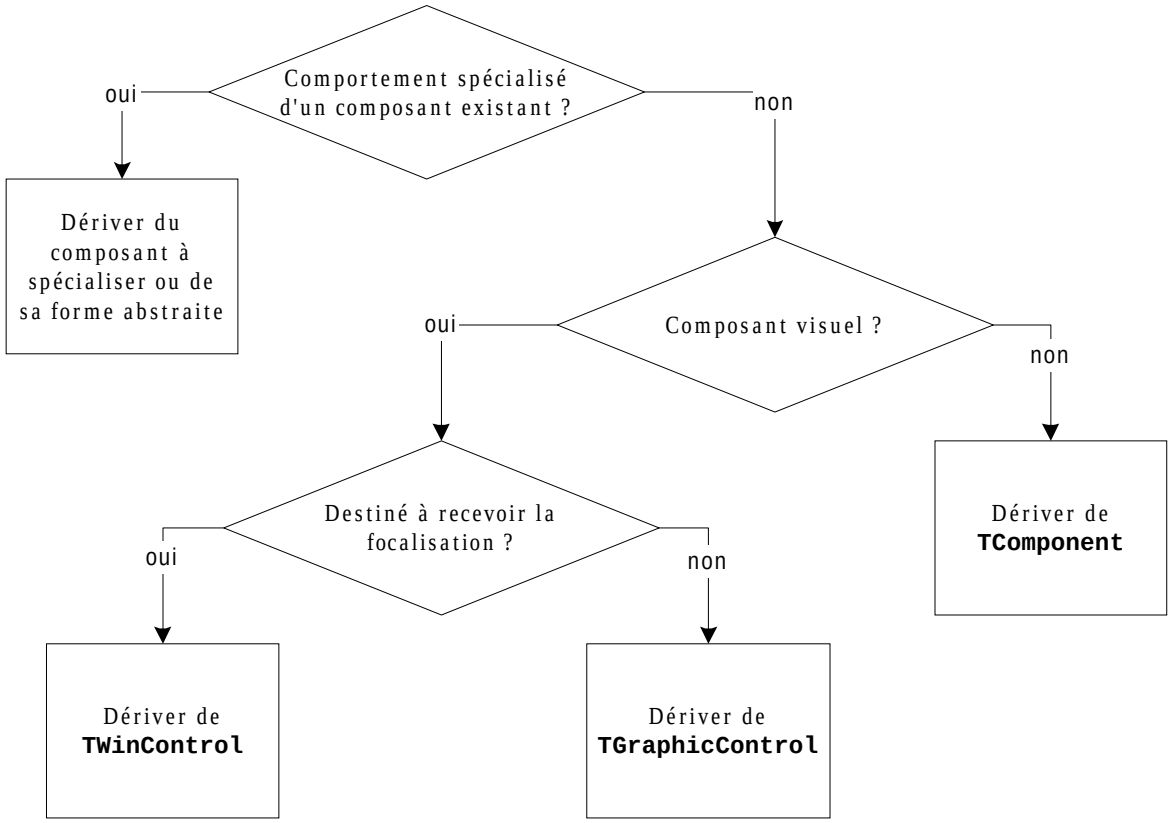


Figure 7.3 détermination de la classe de base d'un nouveau composant

Si vous souhaitez spécialiser ou modifier le comportement d'un composant déjà existant, la réponse est très simple : il suffit de dériver ce dernier. Dans tous les autres cas, la réponse est un peu plus compliquée.

7.2.2.1 Custom ou pas Custom ?

Si vous observez la spécification de la VCL, vous verrez que la plupart des composants directement utilisables dérivent d'un autre composant dont le nom contient en plus le mot **`Custom`**. Par exemple, **`TControl`** dérive de **`TCustomControl`**.

Les classes contenant **Custom** sont l'équivalent des classes de base virtuelles pures dans le sens où elles ne sont pas destinées à être directement instanciées mais laissent ce soin à leurs classes dérivées qui implémentent les comportement qu'elles décrivent dans leur interface.

La plupart des fonctionnalités (propriétés ou méthodes) d'une classe Custom sont protégées. Il vous appartient alors de rendre publiques (ou publiées) uniquement celles que vous souhaitez rendre visibles à l'utilisateur ce qui vous fournit un excellent contrôle sur l'utilisation de la classe.

7.2.2.2 Créer un composant non visuel

Résolvons en premier lieu le cas des composants non visuels, c'est à dire ceux dont le placement sur une fiche n'entraîne que l'affichage d'une icône. Ils peuvent avoir plusieurs rôles. Citons notamment :

La réservation de ressources Windows : c'est en particulier le cas, des boîtes de dialogue communes de Windows qui apparaîtront sur leur forme normale lors de leur invocation par le programme

Les composants n'ayant aucune représentation graphique. Dans cette catégorie se retrouvent, par exemple, les composants permettant de lier votre application à des éléments de bases de données ou des sockets de réseau.

Reposez vous alors la question, « suis-je bien certain que mon composant n'a aucun lien avec l'un de ceux existants ? » si la réponse est « oui », alors il vous faut dériver directement de TComponent, dans tous les autres cas essayez de trouver dans l'arborescence de la VCL un ancêtre convenable qui permette de vous simplifier la tâche.

Rappelons également que si votre composant non visuel est destiné à être déposé sur une fiche, il faudra fournir une icône de représentation.

7.2.2.3 Créer un composant visuel

Les composants visuels sont ceux dont la représentation sur une fiche reflète exactement l'aspect à l'exécution. Bien qu'ils dérivent toujours de **TControl**, on peut les discriminer en deux grandes catégories à l'aide d'un critère simple : le composant doit il être capable de recevoir la focalisation Windows ?

Si, oui, c'est un composant *actif* et le plus simple consiste à les dériver de **TCustomControl** et, à travers ce dernier de **TWinControl**.

A l'opposé, si votre composant n'est pas destiné à recevoir la focalisation, il vaut mieux dériver de **TGraphicControl**. Le principal intérêt est de ne pas consommer trop de ressources. En effet, à l'opposé des **TCustomControl**, les **TGraphicControl** ne dérivent pas de **TWinControl** et donc ne disposent pas d'un **handle** de fenêtre, ce qui permet de préserver ce type de ressources. Par exemple, un **TButton** dérive de **TWinControl** alors qu'un **TToolButton** dérive de **TGraphicControl**.

L'autre différence important réside dans la gestion de l'aspect. En effet, l'apparence générale d'un **TWinControl** est mémorisée par Windows lorsqu'il est masqué, ainsi, lorsqu'il réapparaît au premier plan c'est le système qui le redessine sans appel à **WM_PAINT**. En revanche, les **TGraphicControl** ne sont pas mémorisés, aussi, lorsque leur aspect doit être remis à jour, il y a génération d'un événement **WM_PAINT** lequel doit être intercepté par l'utilisateur. Dans la VCL, cet événement appelle toujours la méthode **Paint** que vous devrez redéfinir pour que votre composant se redessine correctement.

7.3 Exercice résolu : création du composant TListBoxCool

7.3.1 Motivation et déroulement général de l'exercice

Les composants **TListBox** encapsulent les composants boîtes de listes classiques de Windows.

Le mode de sélection de ces composants est régi par deux propriétés :

- lorsque **MultiSelect** est à **false**, la boîte est en mode de sélection simple : un seul élément peut être sélectionné à la fois, son rang est indiqué par la propriété **ItemIndex**. Si aucun élément n'est sélectionné **ItemIndex** vaut -1.

- Lorsque **MultiSelect** est à **true**, la boîte est en mode de sélection multiple si **ExtendedSelect** est à **false** ou en mode de sélection étendue si **ExtendedSelect** est à **true**. La principale différence entre les modes multiple et étendu tient au fait que ce dernier autorise la sélection et l'ajout de plages à une sélection déjà existante. En pratique, il n'est pas de cas où le mode étendu ne soit pas préférable au mode multiple.

Hors, il est particulièrement pénible d'accéder aux index des éléments sélectionnés en mode multiple ou étendu :

- Le nombre d'éléments sélectionnés est stocké dans la propriété **SelCount** (jusqu'ici tout va bien ☺)

- Il faut ensuite vérifier le statut sélectionné de chaque élément de la liste grâce à la propriété indicée **Selected** qui s'utilise comme un tableau sur l'ensemble des éléments (numérotés de 0 à **ItemCount** –1)

L'exemple de code suivant monte la gymnastique nécessaire pour afficher la liste des index des éléments sélectionnés accompagnés des éléments eux mêmes. Les éléments d'une boîte de liste sont stockés dans la propriété **Strings** de sa propriété **Items**.

```
int indiceSelected=0;
for (int compteur=lbc->SelCount-1;compteur>=0;compteur--)
{
  while (!lbc->Selected[indiceSelected])
    indiceSelected++;
  meStd->Lines->Add(IntToStr(indiceSelected)+" "+
```

```
                    lbc->Items->Strings[indiceSelected]);
  indiceSelected++;
}
```

Programme 7.1 Programme d'affichage des index et des éléments sélectionnés d'une boîte de liste

Le résultat visuel est illustré par la figure suivante où vous pourrez vérifier que cela fonctionne !

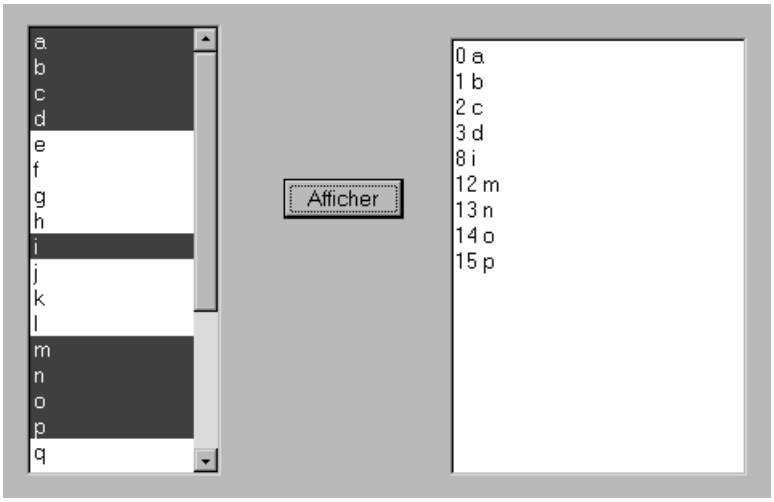


Figure 7.4 Affichage des index et des éléments sélectionnés d'une boîte de liste par le Programme 7.8

Le but est d'ajouter une propriété nommée `Indices` qui permette d'accéder directement aux indices des éléments sélectionnés comme dans l'exemple de code suivant :

```
meCool->Lines->Clear();
for (int compteur=0;compteur < lbc->SelCount;compteur++)
{
  meCool->Lines->Add(IntToStr(lbc->Indices[compteur])+" "+
                     lbc->Items->Strings[lbc->Indices[compteur]]);
}
```

Programme 7.2 Vers une utilisation plus simple des boîtes de listes

Pour résumer, on souhaite que `Indices[i]` nous renvoie le numéro d'index du i^{ème} élément sélectionné.

7.3.2 Mise en place

7.3.2.1 Création d'un nouveau paquet

L'utilisation d'un nouveau paquet simplifie considérablement la gestion des composants car elle a l'avantage de les associer à un projet : celui du paquet. Lorsque vous voudrez modifier votre composant, nous le ferez en chargeant le projet associé à votre paquet (extension **.bpk**).

Voici la marche à suivre :

1. Créez un nouveau paquet (Fichier → Nouveau → Paquet). C'est une opération très simple où l'on ne vous demande que de saisir un nom de fichier et, éventuellement une description comme le montre la figure suivante :

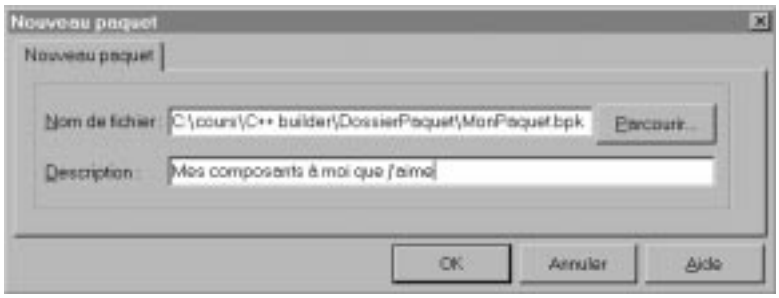


Figure 7.5 Création d'un nouveau paquet

A ce moment là, le paquet est vide. Toutefois, un nouveau projet nommé **`MonPaquet.bkp`** a été créé et il vous est possible de le construire, ce dont vous ne vous privez surtout pas ! (Projet → construire **`MonPaquet`**)

2. La prochaine étape consiste à inclure le paquet nouvellement créé dans la liste des paquets chargés automatiquement par C++ Builder. Ceci se fait grâce à la fenêtre activée par le menu Composant → Installer des paquets.

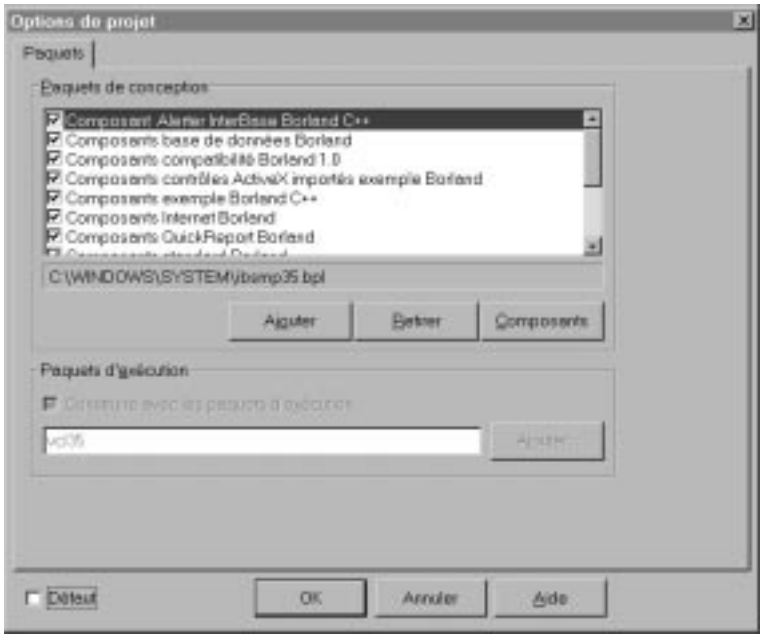


Figure 7.6 le gestionnaire de paquets

Les fonctionnalités de cette boîte de dialogue sont multiples. Tout d'abord, le volet principal vous donne la liste des paquets reconnus par le système. Les boutons Ajouter et Retirer vous permettent respectivement d'ajouter et de supprimer un des paquets. Attention ! retirer ne supprime pas physiquement le fichier contenant le paquet (fichier .bpl) mais seulement sa référence de la liste des paquets reconnus et chargés automatiquement dans la palette !.

Le bouton composants fait apparaître la liste des composants contenus dans un paquet, par exemple, la figure suivante illustre le contenu du paquet « Composants Internet de C++ Builder ».

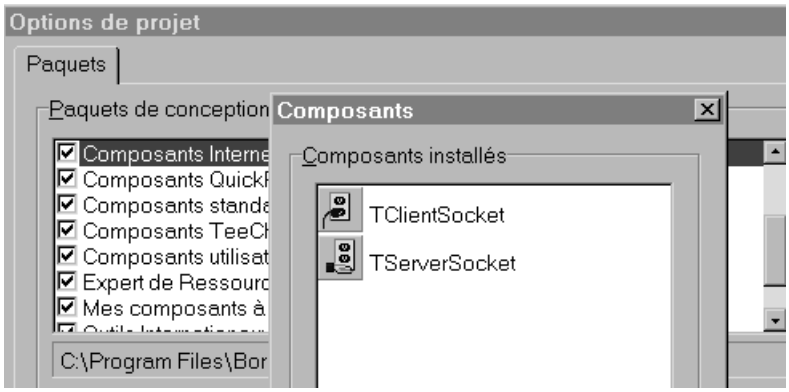


Figure 7.7 Liste des composants d'un paquet

Aussi pratique soit-elle, cette fenêtre ne permet toutefois que de visualiser le contenu d'un paquet. Pour supprimer un composant d'un paquet, il faut en effet éditer le projet et supprimer la référence de l'unité liée au composant comme on le ferait pour supprimer un fichier source du projet d'un exécutable.

3. Utilisez le bouton ajouter pour ajouter votre paquet à la liste des paquets actifs.

Par défaut, un nouveau paquet est uniquement utilisé en conception. En d'autres mots, il est utilisé par l'environnement de C++ Builder 3 lorsque vous ajoutez vos composants sur les fiches. Il évite aussi d'utiliser de trop nombreux fichiers `.h` car le fichier d'inclusion `.bpi` du paquet est utilisé.

Toutefois, lors de la création de l'exécutable le code de vos composants est ajouté intégralement augmentant d'autant la taille de l'exécutable. L'autre solution consiste à ajouter votre paquet à la liste des paquets d'exécution. Ainsi, à l'exécution, le fichier `.bpl` créé sera utilisé en tant que DLL permettant de partager le code de votre composant entre les diverses applications qui l'utilisent. Ce mécanisme est particulièrement agréable à utiliser mais nécessite que le fichier `.bpl` soit placé dans un répertoire du **`PATH`** ou bien dans le répertoire standard des paquets de C++ Builder.

7.3.2.2 Création du composant

- Si vous avez fermé le projet correspondant au paquet, rouvrez ce dernier. En effet, créer le composant alors que vous avez ouvert un projet correspondant à un paquet garantit que le composant sera installé dans ce dernier. En outre, lorsque vous voudrez modifier votre composant, il vous suffira d'ouvrir le projet associé au paquet.

- Dérivez un composant de **`TListBox`**, nommez le **`TlistBoxCool`** (en effet, la gestion des boîtes de liste de cette manière sera nettement plus cool, non ? ☺) , logez le dans la palette **`Exemples`**. Comme vous êtes dans le

projet associé à un paquet, il sera automatiquement logé dans ce dernier (je sais, je me répète, mais dans ce cas là c'est nécessaire)

Les fichiers **TListBoxCool.h** et **TListBoxCool.cpp** (si vous ne les avez pas renommés ☺) sont automatiquement créés pour vous et contiennent le squelette de votre composant. Notez l'utilisation d'un espace de nommage dans le fichier **.cpp** pour isoler la fonction **Register** chargée d'inclure dans le registre les informations concernant votre nouveau composant.

7.3.2.3 Aménagement du composant

L'aménagement du composant **TListBox** vers **TListBoxCool** est des plus simples. En effet, il suffit de lui adjoindre une propriété de type entier indexé que nous allons appeler `Indices`.

Afin de nous simplifier la vie, cette propriété sera du type « sans attribut de stockage ». Lorsque nous tenterons d'accéder à la propriété `Indices`, nous relirons la propriété **Selected**, automatiquement mise à jour par les messages Windows. En outre, nous ne donnerons accès à cette propriété qu'à l'exécution (elle est donc en accès public) et en lecture seulement. Le code de déclaration de la classe **TListBoxCool** est le suivant :

```
//---------------------------------------------------------------------------
#ifndef ListBoxCoolH
#define ListBoxCoolH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------------------------------
class PACKAGE TListBoxCool : public TListBox
{
private:

  // Methode de lecture des indices
  int  __fastcall getIndices(int index);

protected:
public:
    __fastcall TListBoxCool(TComponent* Owner);

    // Creation d'une propriété indexée permettant d'accéder directement aux
    // indices des éléments sélectionnées
    __property  int Indices[int index] = {read=getIndices};

__published:

};
//---------------------------------------------------------------------------
#endif
```

Programme 7.3 Déclaration de la code TListBoxCool

Un petit commentaire s'impose quand à ces déclarations. En effet, une propriété doit être scalaire : il n'est pas possible de renvoyer un tableau en propriété. La parade

consiste à créer une propriété indexée : l'indexation est gérée par la méthode d'accès **`getIndices`** qui prend l'index en paramètre.

Ce comportement est bien accordé à l'esprit des propriétés qui peuvent être modifiées à tout instant par un message Windows.

Le code d'implémentation de la méthode `getIndices` est alors :

```
int __fastcall TListBoxCool::getIndices(int index)
{
  // vérification de l'adéquation aux bornes
  if (index >=0 && index < SelCount)
  {
    int indiceSelected=0;
    // parcours de la liste des éléments sélectionnés
    // jusqu'au moment ou l'on arrive au index_ième
    for (int compteur=0;compteur<=index;compteur++)
    {
      while (!Selected[indiceSelected])
        indiceSelected++;
      indiceSelected++;
    }
    return indiceSelected-1;
  }
  else
    return -1;
}
```

Programme 7.4 Implémentation de la méthode d'accès `getIndices`

A chaque appel sur la propriété `Indices`, on effectue un parcours sur la propriété **`Selected`**, laquelle envoie un message Windows au composant liste, ce qui n'est guère efficace mais garantit la justesse des informations !

Le programme suivant indique comme fonctionne notre nouvelle propriété. Vous y retrouvez également l'utilisation de **`Selected`**, notez la simplification !

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  meStd->Lines->Clear();

  // Utilisation de Selected
  int indiceSelected=0;
  for (int compteur=lbc->SelCount-1;compteur>=0;compteur--)
  {
    while (!lbc->Selected[indiceSelected])
      indiceSelected++;
    meStd->Lines->Add(IntToStr(indiceSelected)+" "+
                      lbc->Items->Strings[indiceSelected]);
    indiceSelected++;
  }

  // Utilisation de Indices
  meCool->Lines->Clear();
  for (int compteur=0;compteur < lbc->SelCount;compteur++)
  {
    meCool->Lines->Add(IntToStr(lbc->Indices[compteur])+" "+
                       lbc->Items->Strings[lbc->Indices[compteur]]);
  }
}
```

Programme 7.5 Utilisation de la propriété Indices

7.3.2.4 Compilation du paquet

Lancez maintenant la compilation du paquet « Ctrl-F9 ». Le message de la figure suivants apparaît pour vous avertir de la reconstruction du paquet. Une fois que vous aurez bien saisi les mécanismes sous-jacents, vous pourrez cocher la case « Ne plus afficher ce message » afin de supprimer cette notification visuelle.

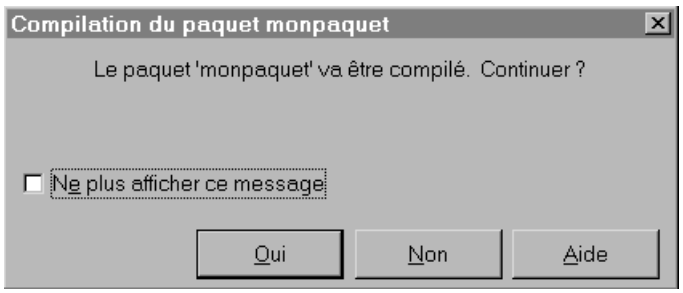


Figure 7.8 Avertissement de recompilation d'un paquet

La première fois que vous compilez un paquet contenant un ou plusieurs nouveaux composants, C++ Builder vous avertit de leur recensement par la fenêtre suivante :

En fait, C++ Builder a non seulement lancé la procédure de compilation et d'édition de lien du paquet mais de surcroît lancé automatiquement l'exécution des fonctions **`Register`** spécialisées dans le recensement des composants.

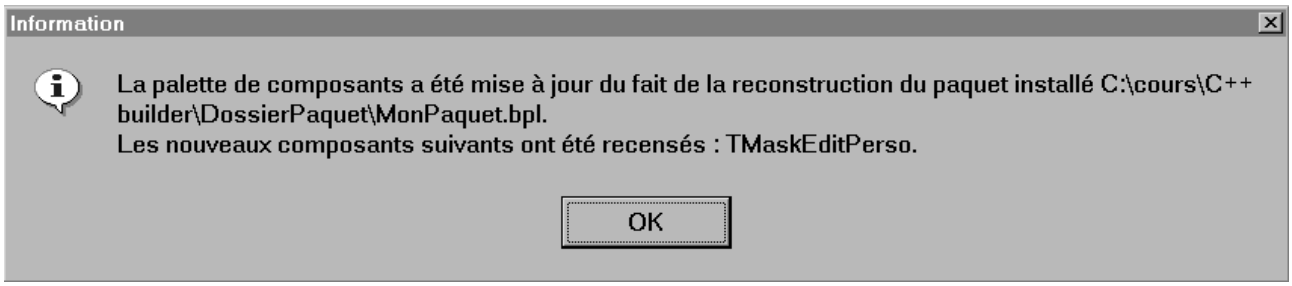


Figure 7.9 Avertissement de mise à jour du registre des composants

Ce message n'apparaît que lors du premier recensement d'un composant. Lorsqu'un composant est connu, les informations qui lui sont relatives sont seulement mises à jour et l'avertissement n'est pas reconduit.

7.3.2.5 Vérification de la palette ...

Vous pouvez dorénavant vérifier que votre composant est présent sur la palette Exemples comme le montre la figure suivante :

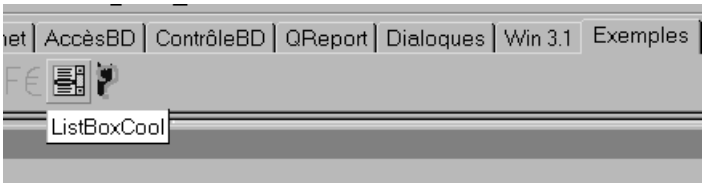


Figure 7.10 Mise à jour de la palette !

Comme nous n'avons pas fourni de nouvelle icône pour ce composant, il reprend celle de son ancêtre le plus récent : **`TListBox`**.

7.3.3 Utilisation du composant

A partir de son installation sur la palette, vous pouvez utiliser votre composant comme bon vous semble et comme s'il s'agissait d'un composant natif de Delphi.

7.3.4 Utilisation d'un composant non installé sur la palette

Il est possible d'utiliser un composant non encore recensé sur la palette. Il s'agit alors de procéder comme s'il s'agissait d'un objet tout ce qu'il y a de plus normal : création manuelle avec appel à **`new`**, affectation des diverses propriétés (n'oubliez pas d'affecter la propriété **`parent`**, sinon le composant ne s'affichera pas …) et destruction à l'aide de **`delete`**. Toutes ces opérations vont être vues plus en détail dans la section suivante dédiée à la création du composant Calculette Francs → Euros.

7.4 Exercice Résolu n°2 : La calculette Francs Euros

Il s'agit ici de créer une calculatrice qui permette de convertir des Francs en Euros et réciproquement. Dans ce cas là, nous nous simplifions l'existence en utilisant le paquet des composants utilisateur : **`dclusr35`**.

Nous allons réaliser un nouveau composant qui en inclut lui même 5 autres :

- Un label indiquant dans quelle boîte saisir / afficher une somme en Francs
- Un label indiquant dans quelle boîte saisir / afficher une somme en Euros
- Un label indiquant le taux de conversion Francs / Euros
- Une boîte de saisie affichant une somme en Francs et où l'utilisateur peut saisir un montant qui sera converti en Euros
- Une boîte de saisie affichant une somme en Euros et où l'utilisateur peut saisir un montant qui sera converti en Francs
- Un bouton qui permettra, dans un second temps, d'invoquer une boîte de dialogue permettant de saisir un nouveau taux pour l'Euro.

Une fois déposé sur une fiche, le résultat final ressemble à la Figure 7.11

Voici le déroulement du projet :

5. Créer le composant
6. Créer des propriétés
7. Gérer l'aspect visuel du composant
8. Gérer les événements internes
9. Associer une icône au composant
10. Ajouter la boîte de saisie du taux de change

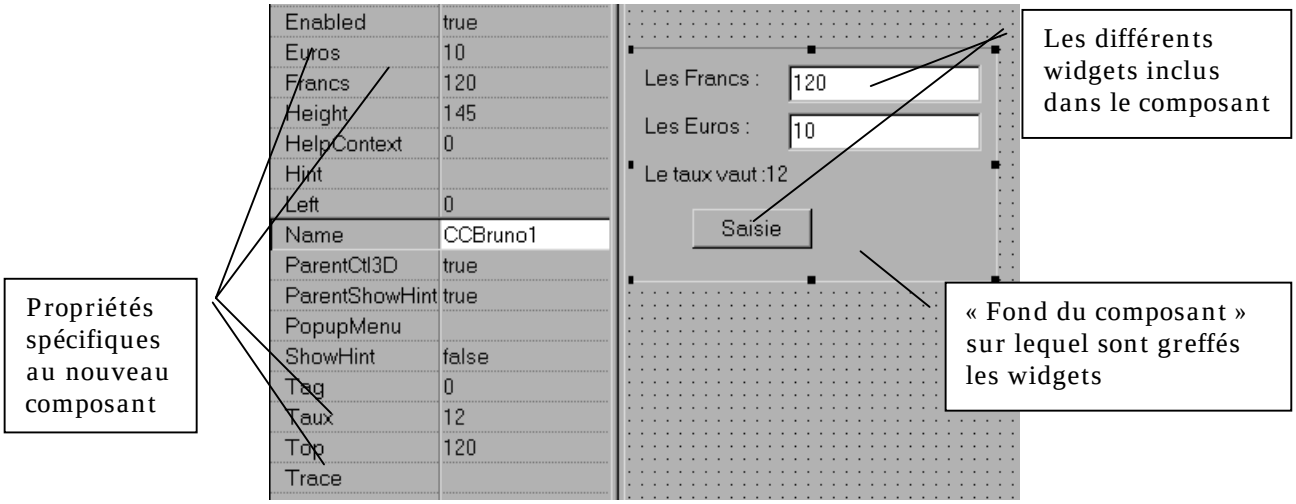


Figure 7.11 Le composant calculette avec ses propriétés associées

7.4.1 Créer le composant

1. Ouvrez le projet associé au paquet des composants exemples. Il se nomme **`dclusr35.bpk`** et se trouve dans le répertoire **`lib`** de l'installation de C++ Builder.

2. La création du composant lui même peut très bien se faire à l'aide de l'expert composant. Le composant Calculette n'étant la version spécialisée d'aucun composant présent sur la palette, nous allons lui donner **`TCustomControl`** comme classe de base (pour plus d'éclaircissements sur le choix de la classe de base, vous pouvez vous reporter à la Figure 7.3).

3. Donnez un nom quelconque au composant, dans toute la suite de cet exposé, le type du composant est `TCCBruno`.

4. Affectez le à la palette « Exemples »

7.4.2 Créer des propriétés

Les propriétés sont très importantes pour un composant car elles permettent de spécifier des valeurs pour les attributs fondamentaux lors de la conception sur fiche. Recensons les propriétés inhérentes au fonctionnement de notre calculatrice :

Taux : Le taux de conversion Francs → Euros

Francs : La somme courante en Francs

Euros : La somme courante en Euros

Chacune de ses propriétés étant fondamentalement de type **double**, nous allons créer trois attributs **private double** qui ne serviront qu'à stocker les valeurs de ces propriétés. Notons immédiatement (avant que quelqu'un n'en fasse la remarque ☺) que deux attributs seulement étaient nécessaires : la valeur des Euros se déduisant de celles des Francs et du Taux et réciproquement. Cette solution est d'ailleurs laissée en exercice.

Par convention, les noms des attributs de stockage des propriétés commencent tous par la lettre F suivie du nom de la propriété, nous obtenons donc :

```
private:
  double FTaux;   // Attribut de stockage du taux de change
  double FFrancs; // Attribut de stockage de la somme en Francs
  double FEuros;  // Attribut (redondant) de stockage de la somme en Euros
```

Programme 7.6 Mise en place d'attributs de stockage des propriétés

Reste à spécifier la politique d'accès aux propriétés. Dans notre cas, le plus simple est d'utiliser des méthodes d'écriture sophistiquées qui assurent la cohérence des données entre elles ainsi que des événements chargés de mettre à jour les valeurs des propriétés lorsque du texte est saisi dans les boîtes d'édition. Ainsi, on peut lire directement la valeur des propriétés dans l'attribut de stockage.

Le reste des définitions inhérentes aux propriétés devient :

```
private:
  __fastcall void SetTaux   (double value);
  __fastcall void SetFrancs (double value);
  __fastcall void SetEuros  (double value);
…
__published:
  __property double Taux   = {read    = FTaux,
                              write   = SetTaux,
                              default = 6,
                              stored  = true};
  __property double Francs = {read    = FFrancs,
                              write   = SetFrancs,
                              default = 0,
                              stored  = true};
  __property double Euros  = {read    = FEuros,
                              write   = SetEuros,
                              default =0};
```

Programme 7.7 Déclaration de méthodes d'accès et des propriétés

Quelques points d'explications :

Les propriétés sont publiées (clause de visibilité **__published**), ce qui permet de les modifier dans l'inspecteur d'objet et leur confère une visibilité de type **public**. En revanche, les méthodes d'accès sont **private**, à l'instar des attributs de stockage

- L'accès en lecture se fait directement sur l'attribut de stockage
- Il est possible de fournir des valeurs par défaut grâce à la clause **`default`** = `valeur`
- Seules les valeurs des propriétés `Francs` et `Taux` sont stockées sur disque (`stored` = **`true`**). En effet, la valeur de `Euros` est immédiatement déductible des deux autres.

7.4.3 Gérer l'aspect visuel du composant

Dans notre cas précis, l'aspect visuel du composant est composé de trois grandes parties :

- Le style du composant, propriété commune à tous les descendants de **`TControl`**.
- Le positionnement et l'affichage de composants inclus (**`TLabel`** et **`TEdit`**)
- La publication de propriétés masquées

7.4.3.1 Le style du composant

L'aspect visuel et comportemental des composants dérivés de **`TControl`** est géré par leur propriété **`ControlStyle`** qui est un **`Set`** d'éléments énumérés. D'après la documentation officielle, les différentes caractéristiques présentes par défaut dépendent fortement de la classe du composant. Voici leur liste exhaustive :

- **`csAcceptsControls`** : Le contrôle devient le parent de tout contrôle placé dedans au moment de la conception. Ce style est activé par défaut pour **`TForm`**, **`TGroup`**, **`TToolBar`**, **`TPanel`** et leurs homologues et désactivé pour les autres.
- **`csCaptureMouse`** :Le contrôle capture les événements de la souris quand on clique dessus. La plupart des contrôles visuels activent cette option.
- **`csDesignInteractive`** : Au moment de la conception, le contrôle redirige les clics du bouton droit de la souris en clics du bouton gauche de la souris manipulant le contrôle.
- **`csClickEvents`** : Le contrôle peut recevoir et répondre aux clics de la souris.
- **`csFramed`** :Le contrôle a un cadre 3D, en creux par défaut (l'inverse d'un **`TPanel`** standard dessiné en relief)
- **`csSetCaption`** :Le contrôle doit changer son libellé pour correspondre à la propriété **`Name`** si le libellé n'a pas été défini explicitement avec une autre valeur. Cette option est particulièrement énervante lorsque l'on travaille en mode conception. Elle est activée par défaut pour tout ce qui est **`TLabel`**,

TGroupBox, **TButton** etc. Dans le cas des **TEdit**, c'est la propriété **Text** qui est modifiée !

- **csOpaque** : Le contrôle remplit entièrement son rectangle client par un rectangle dont la couleur est **clBtnFace** par défaut.

- **csDoubleClicks** : Le contrôle peut recevoir et répondre aux messages de double-clic. Sinon, les doubles clics sont traités comme des simples clics.

- **csFixedWidth** : La largeur du contrôle ne varie pas et ne fait pas l'objet d'une mise à l'échelle, notamment dans le cas du redimensionnement des fenêtres. Typiquement, cette option est toujours activée, il ne faut la désactiver que lorsque vous savez exactement ce que vous faites !

- **csFixedHeight** : Le pendant pour la hauteur de **csFixedWidth** pour la largeur.

- **csNoDesignVisible** : Le contrôle est invisible au moment de la conception. Dans la pratique, cette option n'a que très peu d'intérêt sauf pour les paranoïaques qui fournissent des fiches avec des composants personnalisés qu'ils ne veulent pas faire apparaître. En effet, le propre du composant est tout de même de faciliter la programmation et d'apparaître sur les fiches !

- **csReplicatable** : Le contrôle peut être copié en utilisant la méthode **PaintTo** pour dessiner son image dans un canevas arbitraire. A ma connaissance, cette option est très peu utilisée à l'exception du composant **TShape** dont le but est précisément de définir des « sprites » de conception.

- **csNoStdEvents** : Les événements standard (souris, clavier, clic de la souris) sont ignorés. S'il est possible d'utiliser cet indicateur pour accélérer votre application lorsqu'un composant n'a pas besoin de répondre à ces événements, il est tout de même préférable d'utiliser directement des composants de type gadget ou des formes fixes.

- **csDisplayDragImage** : Le contrôle peut afficher une image extraite d'une liste d'images quand l'utilisateur fait glisser le pointeur de la souris au-dessus du contrôle. C'est typiquement le cas des boutons dans les barres d'outils qui se mettent à clignoter ou changent de couleur de fond.

- **csReflector** : Le contrôle répond aux messages de dialogue Windows, aux messages de focalisation ou aux messages de changement de taille. Utilisez ce paramètre si le contrôle peut être utilisé comme un contrôle ActiveX, de telle sorte qu'il recevra la notification de ces événements.

Sachez également que le constructeur de la classe **TControl** inclut par défaut les options suivants dans **ControlStyle** : **csCaptureMouse**, **csClickEvents**, **csSetCaption** et **csDoubleClicks**. Bien entendu, il vous est toujours possible de modifier ces valeurs, mais ceci doit uniquement être fait dans un constructeur.

Maintenant que nous connaissons la liste des options, intéressons nous au type **Set** qui permet de les manipuler dans la propriété **ControlStyle**.

7.4.3.2 Complément sur l'utilisation des classes Set

Les classes de type **Set** ont été ajoutés au C++ afin de simuler le type **set of** du Pascal utilisé pour modéliser un ensemble non ordonné de valeurs numériques distinctes. De telles constructions sont très pratiques pour modéliser les ensembles d'options, telles que, par exemple, l'ensemble des boutons présents sur une boîte de dialogue de Windows.

Pour simplifier, la déclaration de la classe Set est très similaire à :

```
template<class T, unsigned char valMin, unsigned char valMax>
class Set;
```

Programme 7.8 déclaration simplifiée de la classe Set

où :

- **T** est un type homologue à **unsigned char**, la plupart du temps un type énuméré.
- **valMin** et **valMax** sont les valeurs extrêmes autorisées pour les éléments de l'ensemble.

Du fait de l'utilisation de **valMin** et **valMax** dans le **template**, deux classes basées sur le même type **T** mais de valeurs extrêmes différentes ne sont pas compatibles.

La classe **Set** est pourvue des méthodes présentées dans le tableau suivant. Afin de gagner de la place, le spécificateur **__fastcall**[3] présent pour chacune des méthodes n'est pas présenté.

`Set(void)`	Constructeur par défaut, conçoit un ensemble vide
`Set(const Set&)`	Constructeur par copie
`Set &operator=(const Set&)`	Affectation brutale

`Set &operator *=(const Set& )`
`Set &operator +=(const Set& )`
`Set &operator –=(const Set& )`
	L'ensemble courant devient respectivement l'intersection, la différence ensembliste et la réunion entre lui même et un autre ensemble passé en argument

`bool operator==(const Set&) const` `bool operator!=(const Set&) const`	Comparaisons entre ensembles. Seules les opérations « tous éléments identiques » et son

[3] Ce spécificateur traduit le mécanisme d'appel rapide : les paramètres ne sont pas empilés mais placés dans des registres. Ceci permet de gagner un temps considérable lorsque de nombreux appels sont nécessaires.

	opposé sont fournies. Il n'est pas prévu de pouvoir ordonner des ensembles !
`bool contains(const T) const`	Teste la présence d'un élément dans l'ensemble
`Set & operator<<(const T)`	Ajoute un élément dans l'ensemble
`Set & operator>>(const T)`	Retire un élément de l'ensemble. La classe ne bronche pas si l'on tente de retirer un élément non présent

`Set operator *(const Set& ) const`
`Set operator +(const Set& ) const`
`Set operator –(const Set& ) const`
	Réalisent respectivement l'intersection, la différence ensembliste et la réunion de l'ensemble courant et d'un autre ensemble passé en argument

Tableau 7.1 Les méthodes de la classe `Set`

Vous noterez au passage que les gens de chez Borland n'ont pas du lire Coplien car ils ont programmé de nombreux opérateurs dyadiques en fonction membre ! En outre, ils savent très bien manipuler les fonctions externes comme le prouvent les opérateurs d'entrées / sorties suivants :

```
friend ostream& operator <<(ostream& os, const Set& arg);

friend istream& operator >>(istream& is, Set& arg);
```

Les entrées / sorties sur les ensembles sont assez spéciales dans le sens où elles travaillent sur un mapping de la présence de chaque valeur entre `minVal` et `maxVal` vers [0, 1]. Pour clarifier, si la valeur `i` est présente dans l'ensemble, cela sera signalé par un 1 à l'emplacement `i-minVal` dans la suite de 0 et 1 construite par l'opérateur `<<`. Réciproquement l'opérateur `>>` construit en ensemble à partir d'une suite de 0 et de 1 en ajoutant la valeur correspondant à chaque fois qu'il trouve un 1.

Le code du constructeur du composant Calculette (Programme 7.9) contient l'instruction suivante :

```
(((ControlStyle >> csAcceptsControls)
                  >> csSetCaption)
                  << csFramed)
                  << csOpaque;
```

Nous y apprenons donc que, par rapport au style standard, notre composant :

- N'admet pas de composants enfants (retrait de **`csAcceptsControls`**)
- Ne met pas son **`Caption`** à jour à partir de **`Name`** (retrait de **`csSetCaption`**) ce qui tombe bien car nous n'allons pas lui fournir de propriété **`Caption`**
- Est entouré d'un cadre en creux (ajout de **`csFramed`**)
- Est dessiné de la couleur d'un bouton (ajout de **`csOpaque`**)

Rappelons que par défaut, il accepte les événements de la souris (y compris les doubles clics), et du clavier.

7.4.3.3 Placer les composants inclus

Il n'est malheureusement pas possible d'utiliser l'éditeur de fiche ou un autre éditeur de ressources Windows pour décrire l'emplacement, le titre et autres caractéristiques des composants inclus dans la calculette : toutes ces propriétés doivent être instanciées à la main dans le constructeur de la calculette !

En pratique il est agréable d'utiliser le concepteur pour réaliser une fiche qui a le même aspect que le composant que l'on souhaite créer pour en déduire les valeurs géométriques des composants inclus.

```
__fastcall TCCBruno::TCCBruno(TComponent* Owner) : TCustomControl(Owner)
{

  (((ControlStyle >> csAcceptsControls)
                  >> csSetCaption)
                  << csFramed)
                  << csOpaque;

  Height=200;
  Width=230;

  labelFrancs = new TLabel(this);
  labelFrancs->Top=10;
  labelFrancs->Left=10;
  labelFrancs->Caption="Les Francs :";
  labelFrancs->Parent=this;

  labelEuros = new TLabel(this);
  labelEuros->Top=40;
  labelEuros->Left=10;
  labelEuros->Caption="Les Euros :";
  labelEuros->Parent=this;

  labelTaux = new TLabel(this);
  labelTaux->Top=70;
  labelTaux->Left=10;
  labelTaux->Caption="Le taux vaut : "+FloatToStr(Taux);
  labelTaux->Parent=this;

  saisieFrancs = new TEdit(this);
  saisieFrancs->Top=10;
  saisieFrancs->Left=100;
  saisieFrancs->Text="0";
  saisieFrancs->Parent=this;
  saisieFrancs->Text=FloatToStr(Francs);
  saisieFrancs->OnChange=OnSaisieFrancs;

  saisieEuros = new TEdit(this);
  saisieEuros->Top=40;
  saisieEuros->Left=100;
  saisieEuros->Text="0";
  saisieEuros->Parent=this;
  saisieEuros->OnChange=OnSaisieEuros;
  saisieEuros->Text=FloatToStr(Euros);

  boutonSaisieTaux=new TButton(this);
  boutonSaisieTaux->Top=100;
  boutonSaisieTaux->Left=40;
  boutonSaisieTaux->Caption="Saisie";
  boutonSaisieTaux->OnClick=OnClickBoutonSaisieTaux;
```

Style visuel du composant en cours de création

Ajout des différents composants inclus

Affichage des la propriété Francs

```
  boutonSaisieTaux->Parent=this;
}
```

Programme 7.9 Constructeur du composant Calculette

Ce code d'instanciation est facile à comprendre mais vraiment pénible à taper ! Pensez que vous auriez à réaliser le code entièrement de cette manière si les composants n'étaient pas disponibles sur les palettes et dans l'inspecteur d'objet !

Quelques petites remarques s'imposent :

- Les composants créés ont tous pour propriétaire (**`owner`**) le composant courant (paramètre **`this`** passé au constructeur).
- Le propriétaire est déconnecté du père. En effet, si vous n'affectez pas la propriété **`Parent`**, votre composant ne s'affichera jamais.
- Les caractéristiques géométriques ne sont pas toutes fournies : la largeur et la hauteur des widgets est :
  - la valeur par défaut
  - déduite d'autres caractéristiques, telles que le **`Caption`** d'un **`TLabel`** (auquel s'associe la fonte etc.)

7.4.3.4 Publication des propriétés cachées

La classe **`TCustomControl`** n'est pas destinée à être utilisée directement par les utilisateurs mais bel et bien à être dérivée comme nous le faisons afin de fournir des composants utiles.

A ce titre elle est pourvue d'un nombre impressionnant de propriétés, qui, pour la plupart sont déclarées **`protected`**. Autrement dit, elle n'apparaîtraient pas dans l'inspecteur d'objet des classes dérivées … à moins d'être publiées par l'héritier !

Grâce à ce mécanisme, seules les propriétés utiles pour un composant seront mises à disposition de l'utilisateur. Dans notre exemple, nous avons publié les propriétés suivantes :

```
__published:
  __property Align ;
  __property Ctl3D ;
  __property DragCursor ;
  __property DragMode ;
  __property Enabled ;
  __property ParentCtl3D ;
  __property ParentShowHint ;
  __property PopupMenu ;
  __property ShowHint ;
  __property Visible ;
```

Programme 7.10 Publication des propriétés cachées

En effet, ces propriétés sont assez standard et permettent à l'utilisateur de position facilement son composant (**Align**), de lui apporter des enrichissements (**Ctl3D**, **ParentCtl3D**, **ShowHint**, **ParentShowInt**) et de spécifier dès la conception son aptitude à recevoir des événements (**Enabled**) ou à être visible (**Visible**).

7.4.3.5 Codage des méthodes d'accès en écriture aux propriétés

Maintenant que nous disposons des contrôles d'affichage des propriétés, il nous est possible de spécifier du code pour les méthodes d'accès en écriture des propriétés. En effet, ces dernières ne manqueront pas de mettre à jour l'affichage du composant lorsque les sommes ou le taux change. Notons que pour éviter tout conflit, lorsque le taux change, les sommes en Francs et en Euros sont remises à zéro.

```
void __fastcall  TCCBruno::SetTaux(double value)
{
  // Affectation des valeurs aux propriétés
  FTaux=value;
  FEuros=0;
  FFrancs=0;

  // Mise à jour de l'affichage
  saisieFrancs->Text=FloatToStr(FFrancs);
  saisieEuros->Text=FloatToStr(FEuros);
  labelTaux->Caption="Le taux vaut :"+FloatToStr(FTaux);
}

void __fastcall  TCCBruno::SetFrancs(double value)
{
  // Affectation des valeurs aux propriétés
  FFrancs = value;
  FEuros = FFrancs / FTaux;

  // Mise à jour de l'affichage
  saisieEuros->Text=FloatToStr(FEuros);
  saisieFrancs->Text=FloatToStr(FFrancs);
}

void __fastcall TCCBruno::SetEuros(double value)
{
  // Affectation des valeurs aux propriétés
  FEuros = value;
  FFrancs = FEuros * FTaux;

  // Mise à jour de l'affichage
  saisieFrancs->Text=FloatToStr(FFrancs);
  saisieEuros->Text=FloatToStr(FEuros);
}
```

Programme 7.11 code d'implémentation des méthodes d'accès en écriture aux propriétés "financières"

Quelques remarques s'imposent (mais si ☺) :

- Les méthodes d'accès aux propriétés (que ce soit en lecture ou en écriture) ne sont toujours de type **__fastcall** ce qui implique en particulier qu'elles ne sont jamais **inline**.

En effet, il est important que ce soient de véritables méthodes et non pas du code **inline** car ces méthodes sont susceptibles d'êtres appelées via les mécanismes d'OLE.

Rappelons également que le protocole **__fastcall** remplace l'empilement traditionnel des paramètres par leur placement dans des registres ce qui permet de gagner un temps considérable lors de l'appel, le mécanisme de stack call étant, par définition, assez lent.

- Il faut faire attention à ne pas introduire de mécanisme en boucle lors de l'écriture des méthodes d'accès en écriture. Ainsi, vous remarquerez que les propriétés sont utilisées via leur attribut de stockage dans ce type de code. Ceci est destiné à éviter des mécanismes d'appel infini.

  En effet, supposez que dans la méthode SetFrancs, vous ayez l'instruction :

```
Euros=… ;
```

  Cette ligne aurait pour effet d'invoquer la méthode SetEuros

  Et, si, pour couronner l'ensemble, la méthode SetEuros contient une ligne du style :

```
Francs=… ;
```

  … qui invoque à son tour la méthode SetFrancs, on se retrouve dans un cas de cercle vicieux où chaque méthode passe son temps à invoquer l'autre !

  Moralité : faites bien attention à l'utilisation des propriétés par leurs méthodes d'accès en écriture lorsque vous écrivez l'une d'entre elles !

7.4.4 Gestion des événements internes

Tel quel, notre composant est capable de s'afficher et d'afficher correctement le taux de conversion ainsi que les sommes passées dans ses propriétés lors de la conception. Toutefois, il n'est pas totalement fonctionnel. En effet, il devrait réagir à la frappe de valeurs dans les boîtes d'édition, ce qui revient à mettre en place des méthodes de gestion des événements **OnChange** des composants **TEdit**. Nous parlons ici d'événements *internes* car ils mettent en jeu des composants internes à notre Calculette. Un chapitre ultérieur traitera des événements externes.

La mise en place de gestionnaires d'événements se fait en trois temps :

1. Déclarer une méthode du composant Calculette qui contient le code de gestion

2. Affecter cette méthode à une propriété du composant censé générer l'événement.

3. Implémenter le gestionnaire

Nous allons désormais détailler chacune de ces opérations.

7.4.4.1 Déclaration d'un gestionnaire d'événement

Bien entendu, chaque gestionnaire d'événement a une signature particulière qu'il va falloir respecter. Le meilleur moyen pour connaître la signature d'un événement est encore de consulter l'aide.

Par exemple, si l'on désire gérer les changements de texte d'un **TEdit**, on recherche la rubrique événements du **TEdit** et l'on choisit **OnChange** comme le montre le code suivant :

```
__property Classes::TNotifyEvent OnChange = {read  = FOnChange,
                                             write = FOnChange};
```

Nous y apprenons que les événements sont des propriétés (à accès direct de surcroît). On s'en doutait un peu car il est possible d'associer un gestionnaire à un événement lors de la conception.

Poursuivons notre investigation par la lecture de l'aide associée au type **TNotifyEvent**.

```
typedef void __fastcall (__closure *TNotifyEvent)(System::TObject* Sender);
```

Comme l'on pouvait s'y attendre, il s'agit d'un type pointeur de fonction. Lorsque la propriété est affectée, la méthode référencée par le pointeur est appelée lors du déclenchement de l'événement.

Le mot clef **__closure** est un modificateur interne à C++ Builder qui permet d'affecter des pointeurs de méthodes à une classe différente de celle spécifiée. Cette « magouille » était nécessaire pour simplifier le mécanisme d'affectation des propriétés. Il n'est pas nécessaire de le comprendre pour gérer les événements ☺.

Il suffit de retenir que dans notre cas, la méthode doit avoir le prototype spécifié par le type **TNotifyEvent** à l'exception du **__closure**, soit :

```
void __fastcall NomMethode(TObject *Sender);
```

Programme 7.12 Prototype des événements de type Notification

La plupart des gestionnaires d'événements répondent à cette signature ; en particulier tous les gestionnaires de type **OnClick**. Le paramètre Sender permet d'identifier par son adresse l'objet qui a émis le message. C'est tout particulièrement utile lorsque vous associez le même code à plusieurs objets, par exemple, des boutons radio d'un même groupe.

Il existe toutefois de nombreux autres types de gestionnaires d'événements, en particulier, ceux des événements clavier :

```
void __fastcall OnKeyDown (TObject* Sender, Word &Key, TShiftState Shift);
void __fastcall OnKeyUp   (TObject* Sender, Word &Key, TShiftState Shift);
void __fastcall OnKeyPress(TObject* Sender, char &Key);
```

Programme 7.13 Prototypes des événements associés au clavier

Examinons plus en détail ces événements :

OnKeyDown est généré lorsqu'une touche quelconque du clavier est enfoncée

OnKeyUp est généré lorsqu'une touche quelconque du clavier est relâchée

Leurs paramètres communs sont :

- **Sender** : adresse du composant émetteur de l'événement
- **Key** : code clavier virtuel (géographique ☺) de la touche
- **Shift** : paramètre de type **Set** pouvant contenir trois valeurs associées aux trois modificateurs du clavier : **ssShift**, **ssCtrl** et **ssAlt**. Leur présence dans Shift indique que la touche correspondante du clavier est enfoncée.

OnKeyPress signale qu'une touche alphanumérique a été activée. Ici c'est le caractère correspondant à la touche qui est renvoyé.

Pour conclure cette section, signalons qu'il existe de nombreux événements associés à la souris qui seront plus ou moins disponibles selon le type de composant

OnMouseDown généré lors de l'enfoncement d'un des boutons

OnMouseUp généré lorsque l'utilisateur relâche un bouton (lequel était précédemment enfoncé ☺)

OnClick : gestion d'un cycle enfoncement relâchement complet

OnDblClick : gestion d'un double clic. Notez bien que si vous fournissez une gestion à **OnClick** et **OnDblClick**, et qu'un double clic se produit, **OnClick** sera appelé avant **OnDblClick**. Moralité : l'action associée à un double clic doit prolonger celle associée à un clic simple !

OnMouseMove : gestion des mouvements de la souris.

Une fois les méthodes déclarées avec le bon prototype, il ne reste plus qu'à écrire le code correspondant. Nous donnons ici, à titre d'exemple, le code associé à l'événement **OnChange** de la boîte d'édition de la somme en Francs.

7.4.4.2 Implémentation d'un gestionnaire d'événement

```
void __fastcall TCCBruno::OnSaisieFrancs(TObject *Sender)
{
  // Tentative de conversion du texte de la boîte d'éditions en
  // valeur numérique
  try
  {
    Francs=StrToFloat(saisieFrancs->Text);
  }
```

```
  catch (...)
  {
    // En cas d'échec : affichage d'un message d'erreur
    ShowMessage("Saisie invalide");
  }
}
```

Programme 7.14 implémentation d'un gestionnaire d'événement interne

Bien entendu, ce gestionnaire est simpliste et ne saurait être adapté directement à une utilisation dans un cas réel. Remarquons qu'une grande partie du travail est délégué à la méthode d'accès en écriture de la propriété Francs. Ce comportement est assez généralisé, car il permet d'éviter de la duplication de code. En effet, saisie d'une valeur dans la boîte ou l'affectation d'une valeur par programmation à la propriété Francs sont des fonctionnalités équivalentes qui nécessitent toujours la mise à jour de l'affichage du composant.

7.4.4.3 Affectation d'un gestionnaire à un événement

C'est la partie la plus simple de notre discours : il suffit d'affecter l'adresse de la méthode à la propriété responsable de l'événement. Voir à ce sujet le code du constructeur (Programme 7.9).

7.4.5 Ajouter la saisie du taux

Nous désirons désormais ajouter la possibilité de saisir un nouveau taux de change. Pour cela, l'utilisateur devra appuyer sur le bouton, ce qui entraînera l'apparition d'une boîte de dialogue lui permettant d'effectuer sa saisie.

7.4.5.1 Création de la boîte de dialogue

La boîte de dialogue est créée en tant que fiche ordinaire et présentera l'aspect illustré par la Figure 7.12. Veillez à ce que le projet courant soit celui associé au paquet lorsque vous créez la fiche !

Figure 7.12 Boîte de saisie du taux de l'Euro

7.4.5.2 Utilisation de la boîte de dialogue

Il nous faut maintenant utiliser notre nouvelle boîte de dialogue à l'intérieur de notre composant. Cette étape s'effectue de manière classique à 2 exceptions près.

- Tout d'abord, vous noterez que le menu Fichier → Inclure Entête d'Unité est désactivé lorsque vous éditez le code d'un composant. Aussi il vous faudra ajouter la directive **#include** manuellement, ce qui n'est guère compliqué. Si nous n'avez pas besoin des déclarations contenues dans le Header de la classe Boîte de Dialogue, je vous conseille de tout positionner dans le .cpp du composant Calculette de la manière suivante :

```
#include "CCBruno.h"          // Header du composant Calculette
#include "UTauxEuro.h"        // Header de la boîte de dialogue
#pragma link "UTauxEuro.obj"  // Fichier objet de la boîte de dialogue
```

- Notez la présence de la directive **pragma link** dans le code précédent. Cette dernière force le lieur à incorporer le fichier fourni en argument dans l'exécutable fourni.

  En effet, lorsque vous incorporez un composant dans un projet, par défaut, le lieur n'incorpore dans l'exécutable que le code intrinsèque au composant. Ainsi, dans le cas qui nous intéresse, le code de la boîte de dialogue ne serait pas inclus. La directive **pragma link** pallie ce manque.

  Notons qu'il existait un autre possibilité : ajouter le fichier .cpp de la boîte de dialogue au projet utilisant la Calculette. Cette solution, bien que correcte du point de vue de l'édition de liens souffrait toutefois d'un défaut majeur : elle nécessite de connaître le nom du fichier source de la boîte de dialogue et par là même la structure du composant Calculette alors que tout composant doit être vu comme une boîte noire.

- Chaque unité décrivant une fiche définit une variable externe qui est habituellement créée au lancement de l'application. Lorsque l'on utilise une fiche uniquement dans le cadre d'un composant, cette instance n'est pas créée automatiquement. Aussi, vous devrez le faire vous même, comme dans le fragment de code suivant où vous noterez que nous passons au constructeur de la fiche la variable **Application** comme composant père. Cette variable est définie par toute application VCL.

```
void __fastcall TCCBruno::OnSaisieEuros(TObject *Sender)
{
  int resultat;
  try
  {
    FSaisieTaux = new TFSaisieTaux(Application);
    FSaisieTaux->editTaux->Text=FloatToStr(FTaux);
    resultat=FSaisieTaux->ShowModal();
    if (resultat==mrOk)
    {
      try
      {
        Taux=StrToFloat(FSaisieTaux->editTaux->Text);
        saisieFrancs->Text="0";
        saisieEuros->Text="0";
      }
      // Gestion des erreurs liées à la conversion de la chaîne
      // en nombre réel
      catch (...)
      {
```

```
        if (FMemoTrace)
          FMemoTrace->Lines->Add("Conversion impossible du taux");
      }
    }
  }
  // Récupère les erreurs inhérentes à la création et l'exécution
  // de la boîte de dialogue
  catch (...)
  {
    if (FMemoTrace)
      FMemoTrace->Lines->Add("Erreur lors de la saisie du taux");
  }
  delete FSaisieTaux;
}
```

Programme 7.15 Utilisation de la boîte de dialogue de saisie du taux de l'Euro

7.5 Exercice résolu n°3 réalisation d'un composant de saisie de date

Le but de cet exercice est de réaliser un composant permettant de saisir une date. Ce dernier aura l'aspect visuel suivant :

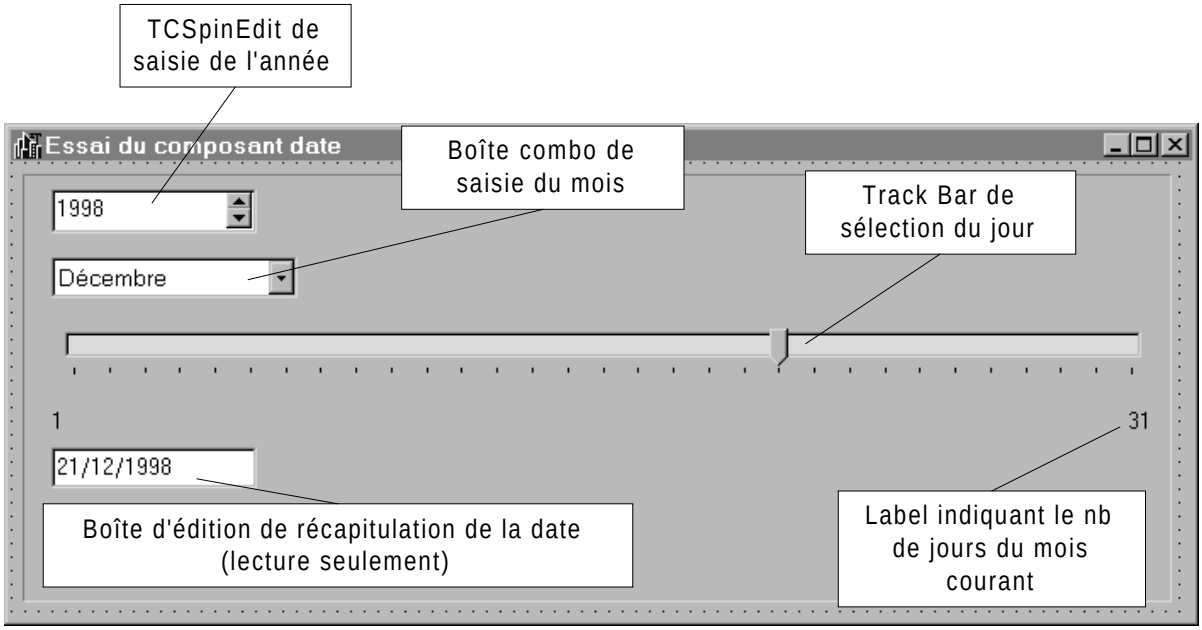


Figure 7.13 Composant de saisie de date

Le composant exporte 4 propriétés : `Annee`, `Mois`, `Jour` et `Date` (lecture seule) qui contiennent respectivement le numéro de l'année indiquée par le composant, le numéro du mois, le numéro du jour ainsi que la date complète compilée sous forme d'une chaîne de caractères.

Comme dans le cas précédent, les méthodes d'accès en écriture aux propriétés sont chargées de mettre à jour les propriétés internes des composants inclus. Afin de faciliter les mises à jour, on utilisera une méthode nommée `ReconstructionDate` et qui sera chargée de mettre à jour la valeur compilée dans la date dans la boîte d'édition idoine.

Le mode de création est similaire à celui du composant calculette Francs / Euros à un détail près : ici il y a un composant Boîte Combo et celui-ci nécessite un traitement préférentiel.

En effet, il serait tentant de vouloir initialiser le contenu de la boîte combo dans le constructeur du composant. Toutefois, ceci est irrémédiablement sanctionné par une erreur de violation de mémoire lorsque l'on tente de poser le composant sur une fiche. Aussi, faut-il passer par une astuce.

Réfléchissons ensemble, quelle est la première méthode qui sera appelée après le constructeur ? bien évidemment, il s'agit de **Paint** afin de mettre à jour l'aspect visuel du contrôle !

Aussi, il nous sera possible de procéder à des initialisations dans la méthode **Paint**. Toutefois, nous ne voulons procéder à ces initialisations que lors de la première invocation de **Paint**, il était possible d'ajouter un attribut booléen servant de drapeau dans la définition de la classe mais nous préférons utiliser une astuce : juste après sa construction, la boîte combo ne contient aucun élément ! aussi, nous n'entrons dans ce code d'initialisation que si cette condition est vérifiée.

De manière générale, il sera impossible de toucher aux propriétés tableau dans le constructeur du composant. Ceci est spécialement vrai pour toutes les propriétés nommées **Lines** ou **Items**.

En outre, le code d'initialisation du composant à la date courante (ou une date quelconque) qui ne pouvait être fait dans le constructeur toujours à cause de cette maudite boîte combo, est également déplacé dans **Paint**. Vous noterez au passage que nous utilisons allègrement le fait que **Paint** est également appelée lorsque l'on dépose un composant sur une fiche.

Finalement, les codes du constructeur et de **Paint** deviennent :

```
__fastcall TBrunoDate::TBrunoDate(TComponent* Owner)
                                  : TCustomControl(Owner)
{
  // Style du contrôle
  (((ControlStyle >> csAcceptsControls)
                  >> csSetCaption)
                  << csFramed)
                  << csOpaque;

  // Propriétés géométriques
  Width=675;
  Height=250;

  // Construction des différents contrôles

  editAnnee=new TCSpinEdit(this);
  editAnnee->Parent=this;
  editAnnee->Left=16;
  editAnnee->Top=8;
  editAnnee->MinValue=1980;
  editAnnee->MaxValue=2100;
  editAnnee->OnChange=OnChangeAnnee;

  editDate=new TEdit(this);
```

```
  editDate->Parent=this;
  editDate->Left=16;
  editDate->Top=160;

  // On construit normalement la boîte combo
  // Il faut néanmoins se garder de toucher à la propriété Items
  comboMois=new TComboBox(this);
  comboMois->Parent=this;
  comboMois->Left=16;
  comboMois->Top=48;
  comboMois->Style=csDropDownList;
  comboMois->OnChange=OnChangeMois;

  tbJour=new TTrackBar(this);
  tbJour->Parent=this;
  tbJour->Left=16;
  tbJour->Top=88;
  tbJour->Width=650;
  tbJour->Min=1;
  tbJour->Max=31;
  tbJour->OnChange=OnChangeJour;

  labelPremier=new TLabel(this);
  labelPremier->Parent=this;
  labelPremier->Left=16;
  labelPremier->Top=136;
  labelPremier->Caption="1";

  labelDernier=new TLabel(this);
  labelDernier->Parent=this;
  labelDernier->Left=650;
  labelDernier->Top=136;

}
```

Programme 7.16 Code du constructeur

```
void __fastcall TBrunoDate::Paint(void)
{
  TCustomControl::Paint();             // Appel de Paint du parent

  if (comboMois->Items->Count==0)      // Test sur le nb d'éléments de
                                       // la boîte combo
  {
    // Remplissage de la boîte combo
    comboMois->Items->Add("Janvier");
    comboMois->Items->Add("Février");
    comboMois->Items->Add("Mars");
    comboMois->Items->Add("Avril");
    comboMois->Items->Add("Mai");
    comboMois->Items->Add("Juin");
    comboMois->Items->Add("Juillet");
    comboMois->Items->Add("Aout");
    comboMois->Items->Add("Septembre");
    comboMois->Items->Add("Octobre");
    comboMois->Items->Add("Novembre");
    comboMois->Items->Add("Décembre");

    // Récupération de la date courante par un appel système Windows

    SYSTEMTIME  toutDeSuite;
    GetLocalTime(&toutDeSuite);

    // Modification des propriétés du composant en conséquence
```

```
    Annee=toutDeSuite.wYear;
    Mois=toutDeSuite.wMonth;
    Jour=toutDeSuite.wDay;
  }
  ReconstructionDate(); // Mise à jour de la boîte d'édition
}
```

Programme 7.17 Code de Paint

7.6 Gestion des événements externes

Au paragraphe 7.4.4 nous avons vu comment gérer des événements internes à notre composant. Nous allons désormais nous intéresser aux événements *externes* c'est à dire les événements fournis par notre composant Calculette à l'utilisation.

7.6.1 Motivation et mise en œuvre

Il est important de fournir des événements au programmeur qui utilise votre composant. Le choix des événements est assez délicat. En effet, les événements que vous publiez doivent faciliter les opérations suivantes :

- Obtenir des indications (on parle habituellement de *notifications*) sur les opérations subies par le composant, par exemple, l'utilisateur à modifié la date courante

- Interagir avec l'utilisateur final du produit en lui permettant d'ajouter ses propres commandes en réponse à des événements standard.

La première chose à savoir est qu'un événement est avant tout une propriété de type **`closure`**. Comme nous l'avons vu un peu plus haut, ce type est associé à des pointeurs de méthodes dans un objet fiche.

Il existe deux grands types d'événements externes :

- Ceux qui sont déjà prévus par C++ Builder mais qui sont masqués, il suffira de les rendre accessibles par le mécanisme de publication des propriétés **`protected`** que nous avons déjà utilisé afin de rendre visibles les propriétés standard des contrôles.

- Les événements que vous souhaiterez ajouter vous même.

Que vous publiiez un événement standard ou personnalisé, il apparaîtra toujours dans la page événements de l'inspecteur d'objet.

7.6.2 Un exemple simple

Nous allons ajouter trois événements utilisateur à notre composant de saisie de date.

- Le premier consistera à rendre visible l'événement standard **`OnClick`**

- Les deux autres sont spécifiques au composant de saisie de date et seront respectivement émis avant et après les opérations de changement de date.

Afin de rendre les choses simples, ces deux derniers événements seront de type notification : c'est à dire avec un seul paramètre correspondant à l'émetteur du message. On se réfèrera au chapitre pour de plus amples renseignements sur les notifications.

Il faut également penser à donner des noms éloquents aux événements. En effet, ils doivent clairement indiquer dans quelles circonstances ils seront invoqués. En outre, la tradition veut que leur nom commence systématiquement par **On**. Suivant ces recommandations, nous appellerons nos événements `OnAvantChangementDate` et `OnApresChangementDate`.

Leur déclaration obéit au code suivant :

```
class PACKAGE TBrunoDate : public TCustomControl
{

  ...

  // Attributs de stockage des propriétés associées aux événements spécifiques

  TNotifyEvent FAvantChangementDate;
  TNotifyEvent FApresChangementDate;

  ...
__published:

  // Publication de OnClick auparavant en protected
  __property OnClick;

  // Publication des deux événements spécifiques
  __property TNotifyEvent OnAvantChangementDate= read = FAvantChangementDate,
                                                 write = FAvantChangementDate};
  __property TNotifyEvent OnApresChangementDate={read = FApresChangementDate,
                                                 write = FApresChangementDate};
  ...
};
```

Programme 7.18 Déclaration et publication d'événements utilisateur

Une fois ces déclarations réalisées, les événements apparaissent dans l'inspecteur d'objet (comme le montre la figure suivante) et se manipulent exactement de la même manière que les événements associés aux composants standard.

A titre indicatif, le code d'événements très simple est fourni juste en dessous de la figure, les différentes affichages de la date permettront de vérifier que l'événement `OnAvantChangementDate` a bien lieu avant les modifications effectives des propriétés alors que `OnApresChangementDate` a lieu après.

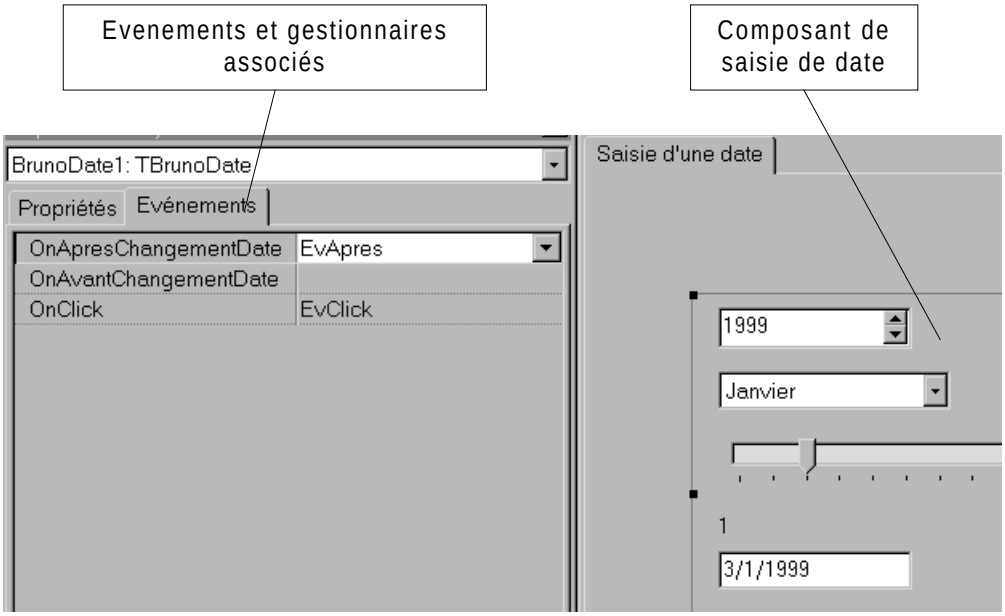


Figure 7.14 Manipulation des événements externes du composant de saisie de date

```
void __fastcall TFPrinci::EvAvant(TObject *Sender)
{
  memoTrace->Lines->Add("Attention : la date va changer ! " +
                        IntToStr(BrunoDate1>Jour)+"/" +
                        IntToStr(BrunoDate1->Mois)+"/" +
                        IntToStr(BrunoDate1->Annee));
}
//---------------------------------------------------------------------------
void __fastcall TFPrinci::EvApres(TObject *Sender)
{
  memoTrace->Lines->Add("Attention : la date a changé ! " +
                        IntToStr(BrunoDate1>Jour)+"/" +
                        IntToStr(BrunoDate1->Mois)+"/" +
                        IntToStr(BrunoDate1->Annee));
}
//---------------------------------------------------------------------------
void __fastcall TFPrinci::EvClick(TObject *Sender)
{
  memoTrace->Lines->Add("Attention : l'utilisateur clique sur le composant");
}
```

Programme 7.19 Code associé aux événements externes du composant de saisie de date

7.6.3 Prendre en compte les événements

Maintenant que nous avons permis à l'utilisateur de fournir ses propres gestionnaires d'événements, il faut en tenir compte et ce en appelant le gestionnaire fourni par l'utilisateur au moyen de la propriété.

Il n'y a jamais à se préoccuper des événements standard genre **`OnClick`**. En effet, par défaut ils appellent des méthodes standard (**`Click`** pour **`OnClick`**) que vous avez toutefois la possibilité de surcharger.

Pour ce qui est des événements spécifiques, il vous appartient de les activer au bon moment en appliquant à la lettre une règle d'or :

Aucune hypothèse ne doit être faite sur le contenu du gestionnaire fourni par votre utilisateur.

En particulier, votre code doit fonctionner que l'utilisateur fournisse ou non un gestionnaire pour votre événement. Aussi, le gestionnaire utilisateur se rajoute au code par défaut et ne doit jamais le remplacer. L'appel à un gestionnaire utilisateur doit ressembler à :

```
if (Propriete)
  Propriete(arguments)
```

Programme 7.20 Code générique de prise en compte d'un gestionnaire utilisateur

La propriété contenant le gestionnaire est à 0 si celui-ci est vide, dans un deuxième temps, vous utilisez la propriété comme tout pointeur de méthode pour appeler le code de votre utilisateur en lui fournissant les bons paramètres.

Dans notre cas, il suffit d'appeler les gestionnaires dans chaque méthode associée aux événements internes de changement des composants comme le montre le code suivant :

```
void __fastcall TBrunoDate::OnChangeAnnee(TObject *Sender)
{
  if (OnAvantChangementDate)
    OnAvantChangementDate(this);
  Annee=editAnnee->Value;
  if (OnApresChangementDate)
    OnApresChangementDate(this);
}

void __fastcall TBrunoDate::OnChangeMois(TObject *Sender)
{
  if (OnAvantChangementDate)
    OnAvantChangementDate(this);
  Mois=comboMois->ItemIndex+1;
  if (OnApresChangementDate)
    OnApresChangementDate(this);
}

void __fastcall TBrunoDate::OnChangeJour(TObject *Sender)
{
  if (OnAvantChangementDate)
    OnAvantChangementDate(this);
  Jour=tbJour->Position;
  if (OnApresChangementDate)
    OnApresChangementDate(this);
}
```

Programme 7.21 Prise encompte des événements spécifiques

Il est possible d'interagir avec le code de votre utilisateur en utilisant des paramètres en entrée sortie comme le prouve l'exercice suivant :

7.6.4 Exercice : interdire la modification de la date (✶✶)

Utilisez un argument booléen pour le gestionnaire `OnAvantChangementDate` afin de pouvoir interdire la modification de la date. On pourra se référer aux événements **`OnBeforeClose`**.

7.7 Conversion d'une boîte de dialogue générique en composant

La manipulation des boîtes de dialogue standard de Windows au travers de composants est particulièrement pratique. En effet, elle permet de positionner facilement des propriétés avant exécution du dialogue, de récupérer le code de retour modal ainsi que des données d'exécution dans des propriétés.

7.7.1 Motivation

Dans cette section, nous allons montrer comment encapsuler une boîte de dialogue dans un composant. Pour cela, nous allons nous appuyer sur l'exemple d'une boîte de saisie générique.

En effet, la plupart des boîtes de dialogue orientées saisie obéissent à la structure suivante :

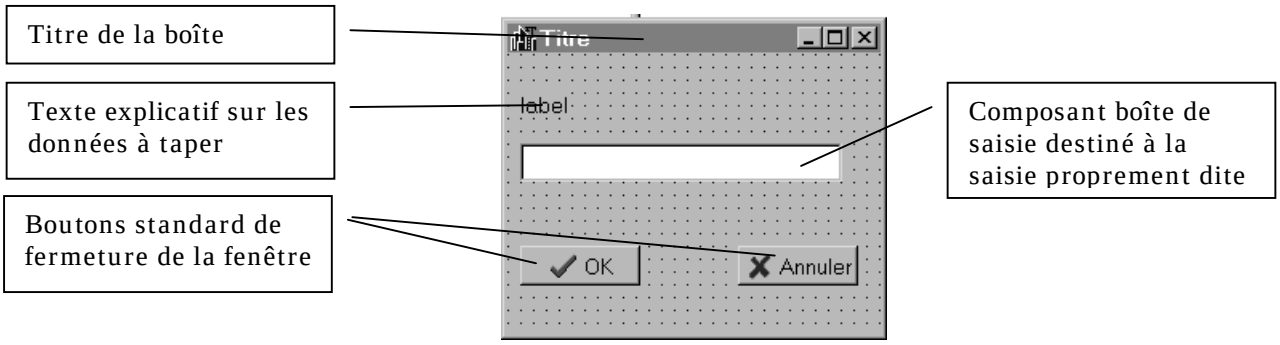


Figure 7.15 Boîte de dialogue de saisie générique

L'encapsulation dans un composant présente les avantages suivants :

Manipulation facilitée : il suffira de déposer une icône depuis la palette de C++ Builder sur une fiche pour utiliser une instance de la boîte de dialogue.

Possibilité de spécifier des propriétés : C'est sans doute le plus important car il permettra de personnaliser (je serai tenter de dire customiser ☺) facilement chaque instance de notre composant, le rendant ainsi agréable à utiliser. Prenons le cas de notre dialogue de saisie, les éléments qui devront être modifiés sont les suivants :

- Le titre de la fenêtre : propriété **`Caption`** de la boîte de dialogue
- Le texte explicatif : propriété **`Caption`** d'un composant **`Label`** de la boîte de dialogue

- Le texte initial de la boîte de saisie : propriété **`Text`** d'un composant **`Edit`** de la boîte de dialogue. C'est également au travers de ce dernier que vous pourrez récupérer le résultat de votre saisie.

S'il est possible de réaliser toutes ces opérations relativement facilement, vous voyez qu'elles impliquent tout de même d'utiliser des propriétés de composants de la boîte ce qui impose de connaître la structure du dialogue.

En créant des propriétés d'un composant englobant cette boîte, vous n'auriez qu'à connaître ceux-ci. Nous allons donc créer 3 propriétés de type texte pour notre dialogue :

- **`Caption`** : associé au titre de la boîte de dialogue
- **`Label`** : associé au texte explicatif
- **`Text`** : associé au texte de la boîte de saisie elle même.

Vous noterez que nous avons respecté les conventions habituelles de la VCL sur les noms des propriétés de texte. La figure suivante récapitule cet état de fait.

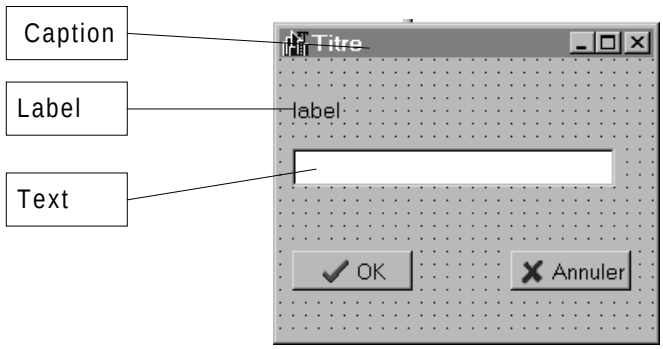


Figure 7.16 les propriétés de notre composant saisie

7.7.2 Fonctionnement de l'encapsulation

Considérez les composants associés aux boîtes de dialogue standard de Windows. Ils sont tous munis de la méthode **`bool __fastcall Execute()`**. C'est précisément elle qui réalise la plupart des opérations :

1. Création de la boîte de dialogue sous jacente
2. Mise en place de son aspect visuel grâce aux propriétés du composant
3. Affichage modal de la boîte de dialogue
4. Récupération du statut de retour modal
5. Modification des propriétés du composant en fonction des actions de l'utilisateur sur la boîte
6. Destruction de la boîte de dialogue et libération des ressources

Il est très important de noter que la boîte de dialogue sous jacente *n'est pas active lors de la conception* sur fiche. Le lien avec les propriétés du composant n'est réalisé que lors de l'activation de la méthode `Execute`.

7.7.3 Réalisation

La première opération consiste à créer la fiche générique comme n'importe quelle fiche. Veillez toutefois à l'ajouter au projet dans lequel vous allez créer votre composant. Dans notre cas, nous l'ajoutons au paquet des exemples et nous nommons l'unité `USaisieGenerale`.

Comme d'ordinaire, il convient de créer le composant que nous allons ajouter à la palette et au paquet des exemples pour raison de simplicité. Bien que vous ayez désormais l'habitude de cette opération, reprenons ensemble le *modus operandi*.

- Ouvrez le projet associé au paquet des exemples (`dclusr35.bpk`)
- Lancez l'expert composant
  - Dérivez de **`TComponent`**
  - Choisissez la palette exemples
  - Le nom de classe utilisé dans l'exemple est `TComposantSaisie`

Comme dans le cas précédent d'utilisation d'une boîte de dialogue par le composant calculette, il est nécessaire d'inclure manuellement l'entête de la boîte de dialogue et les informations d'édition de lien dans le fichier `.cpp` associé à votre nouveau composant. On retrouvera, par exemple, les lignes :

```
#include "ComposantSaisie.h"
#include "USaisieGenerale.h"
#pragma link "USaisieGenerale.obj"
```

Programme 7.22 Mise en place du composant

7.7.4 Mise en place des propriétés

Les propriétés que nous utilisons sont toutes des chaînes de caractères – donc du type **`AnsiString`**. En outre ce sont des propriétés en accès direct ; en effet, rappelez vous (section 7.7.2) que la boîte de dialogue n'est construite qu'au moment de l'appel de la méthode `Execute`, laquelle est chargée de la mise en place des propriétés, aussi, dans notre cas, il n'est point utile de créer des méthodes d'accès compliquées. Le code obtenu est le suivant :

```
private:
  AnsiString FTitre;
  AnsiString FLabel;
  AnsiString FTexte;

__published:
  __property AnsiString Caption = {read=FTitre, write=FTitre, stored=true};
```

```
__property AnsiString Label = {read=FLabel, write=FLabel, stored=true};
__property AnsiString Text = {read=FTexte, write=FTexte, stored=true};
```

Programme 7.23 Création des propriétés du nouveau composant

7.7.5 Codage de la méthode Execute

Comme nous l'avons vu au dessus, la méthode Execute est au centre du processus. Rappelons succinctement qu'elle va créer la boîte de dialogue, mettre en place l'aspect visuel au travers de ses propriétés, afficher la boîte en mode modal, récupérer le code de sortie, mettre à jour ses propriétés en fonction des actions de l'utilisateur avant de détruire la boîte rendant ainsi de précieuses ressources au système.

En outre, elle est souvent responsable des erreurs qui pourraient se produire et gère les exceptions. Le code suivant traduit cet état de fait :

```
bool __fastcall TComposantSaisie::Execute(void)
{
  bool resultat;

  // Creation de la boîte, notez que le parent est l'application
  //  elle même

  TFSaisieGenerale *boite=new TFSaisieGenerale(Application);

  // Mise en place des propriétés

  boite->Caption=FTitre;
  boite->labelEdition->Caption=FLabel;
  boite->boiteEdition->Text=FTexte;

  // Execution modale et récupération du résultat
  resultat=(boite->ShowModal()==mrOk);

  // Mise à jour de la propriété liée à la saisie
  FTexte=boite->boiteEdition->Text;

  // Destruction et libération des ressources
  delete boite;
  return resultat;
}
```

Programme 7.24 la méthode Execute

7.8 Réalisation d'un composant graphique

Le problème est ici totalement différent car l'on ne construit plus un composant capable de recevoir la focalisation. Pour parler plus technique, il ne s'agit plus d'un Widget mais bel et bien d'un Gadget.

La première différence tient à la classe de base. En effet, on ne dérivera plus de **TCustomControl** mais de **TGraphicControl**. Notons bien qu'il était tout à fait possible de dériver de **TCustomControl**, mais il faut l'éviter si le composant n'a pas besoin de recevoir la focalisation du clavier afin de minimiser le nombre de ressources

Windows utilisées (on peut revoir la section 7.2.2 pour une plus ample discussion à ce sujet).

La principale préoccupation du concepteur de composant graphique réside dans l'aspect visuel de ce dernier. En effet, contrairement aux composants fenêtrés dont l'état est sauvegardé par le système, les composants graphiques ne peuvent compter que sur leur méthode virtuelle **`Paint`** (automatiquement appelée par l'événement Windows **`ON_PAINT`**) ainsi que sur un éventuel gestionnaire d'événement **`OnPaint`** fourni par l'utilisateur pour mettre à jour leur aspect visuel après, par exemple, un redimensionnement de leur zone client ou leur retour au premier plan.

7.8.1 Le composant Affichage de Graphe

Le composant graphique que nous nous proposons de réaliser va permettre d'afficher sommairement un graphe à partir du nom d'un fichier passé en propriété.

Les sommets sont matérialisés par un rond bleu et les arcs (non orientés) par un trait vert.

La figure suivante indique le format d'un tel fichier :

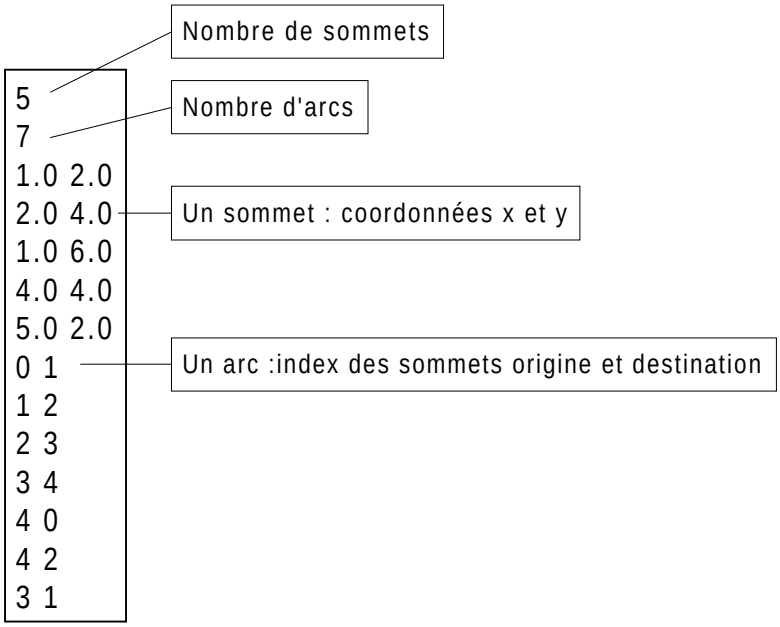


Figure 7.17 Format d'un fichier de graphe

7.8.2 Mise en œuvre

Nous allons proposer les propriétés suivantes :

- Le nombre d'arcs et le nombre de sommets : entiers (`NbSommets` et `NbArcs`) en lecture seulement
- La marge d'affichage : espace en pixels (`Marge`) entre la frontière du composant et l'enveloppe convexe du graphe. C'est un entier en lecture et

écriture. La marge est mise en évidence sur la figure suivante ou le composant `Graphe` occupe la zone client d'un **`TPanel`**.

- Le nom du fichier qui contient le graphe : c'est une chaîne ANSI en lecture et écriture (`Fichier`).

Bien entendu, la majeure partie du travail est accomplie par 2 méthodes :

- L'affectation d'un nom de fichier (partie **`write`** de la propriété `Fichier`) qui alloue de l'espace mémoire pour stocker les informations relatives au graphe (coordonnées des sommets, origine et destination de chaque arc)

- La méthode **`Paint`** d'affichage du composant qui est chargée de mettre à jour l'aspect visuel du composant

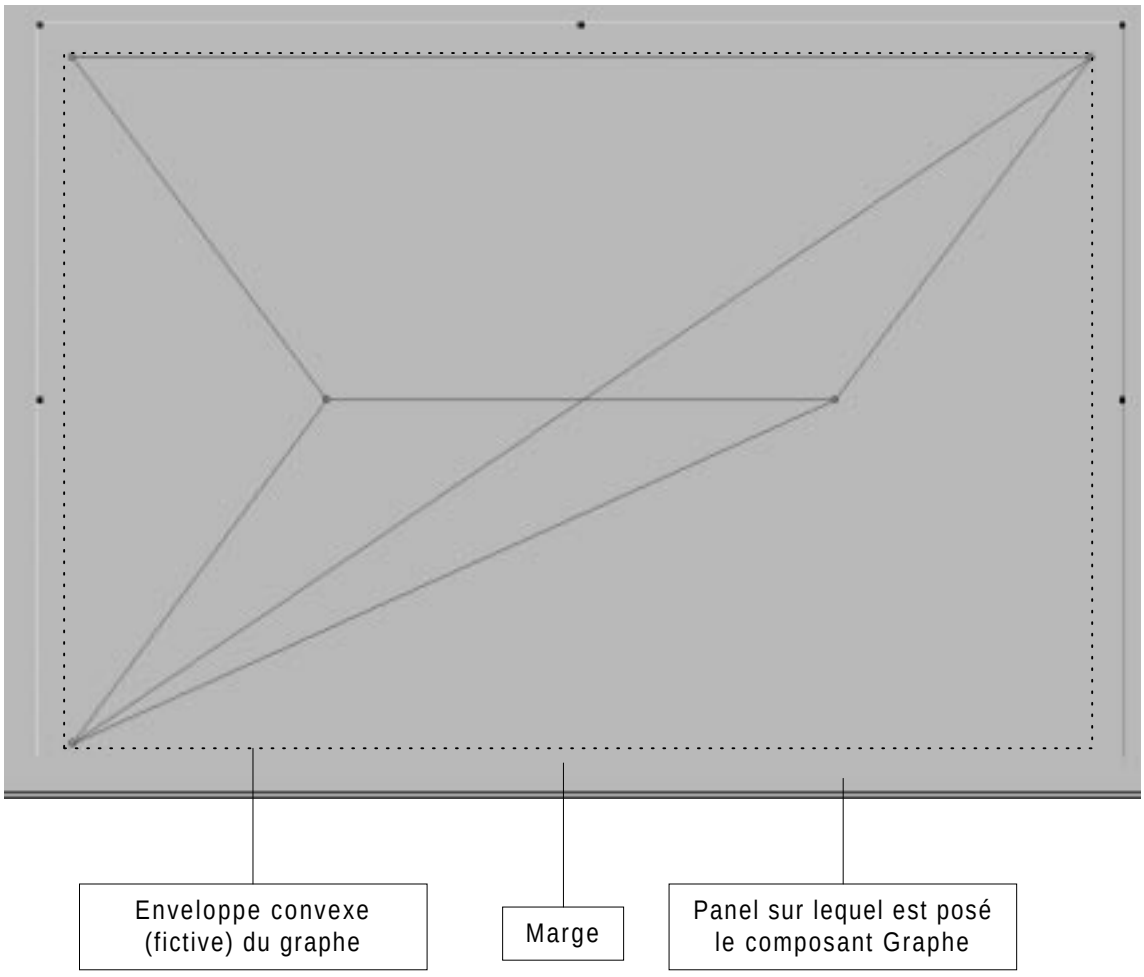


Figure 7.18 Visualisation du composant Graphe

7.9 Exercices supplémentaires

1. Modifier le composant `TListBoxCool` afin que la propriété Indices fonctionne également en écriture. On distinguera le cas `index < SelCount` : un élément sélectionné est remplacé par un autre du cas `index = SelCount` un élément est ajouté à la sélection. Quelle autre modification est elle nécessaire ? (✶✶)

2. Modifier le composant `TListBoxCool` pour qu'il utilise un composant de stockage pour la propriété `Indices`. Celui-ci devra être mis à jour lorsqu'un utilisateur clique sur la liste. Indication : il faudra redéfinir la méthode **`WndProc`**. (✶✶)

3. Modifier le composant Calculette pour qu'il n'utilise pas d'attribut de stockage pour la propriété `Euros`. (✶)

4. Modifier le composant Calculette pour qu'il utilise `TComposantSaisie` en lieu et place de sa boîte dédiée. Est il toujours nécessaire d'utiliser un **`pragma link`** ? (✶)

5. Modifier la boîte de saisie `TSaisieGenerale` et le composant `TComposantSaisie` pour qu'ils intègrent la notion de masque de saisie et traitent les exceptions qui y sont liées (✶✶)

8. En guise de conclusion

J'espère que ce poly (qui ne donne, rappelons-le, qu'un bref aperçu des nombreuses possibilités offertes par C++ Builder) vous donnera envie d'exploiter à fond ce logiciel que, personnellement, j'apprécie beaucoup – bien que regrettant certaines des fonctionnalités offertes par son ancêtre Borland C++. Il ne fait aucun doute qu'au bout de 6 mois de travail intensif sur cet outil, c'est vous qui viendrez me donner des cours !

Tout retour est évidemment le bienvenu et je vous encourage à me l'envoyer par mel : bruno.garcia@lemel.fr ou bruno.garcia@francemel.com .

En outre, j'aimerais dédicacer ce travail à la promo 1999 de F2 de l'ISIMA qui a du se coltiner Borland C++ / OWL en 2ème année puis C++ Builder en 3ème année. Leur compétence et leur curiosité m'a obligé à me remettre en cause en permanence et fait du cours de C++ Builder un défi permanent (et responsable de nombreuses veillées tardives). Finalement, leur gentillesse et leur humour a rendu ce cours particulièrement agréable – quelques épisodes resteront définitivement gravés dans ma mémoire. Aussi, pour toutes ces raisons, merci, merci Florence, Ashwin, Cedrick, Chness, Cyril, Didou, Ecom, Elkine, Fonzy, Fred, Fufu, Goontar, Izno, JD, Jérôme, Jesz, Kal, Laurent, Oumph, Rémi, Roubech, Serge, Totov et Ubik.