Formation C# – Delphi.NET – Delphi Win32
Développement & Sous-traitance

© Copyright 2005 Olivier DAHAN
Reproduction, utilisation et diffusion interdites sans l'autorisation de l'auteur. Pour plus d'information contacter odahan@e-naxos.com

Packages, plugins et réflexion

Les *packages* sous Delphi .NET qui ont beaucoup changé tout en restant les mêmes. Ils sont désormais totalement natifs et permettent de partager des classes avec d'autres langages de la plate-forme. La technique des *plugins* est une façon originale de se servir des packages, c'est aussi une façon moderne d'ouvrir un logiciel à des apports extérieurs. Enfin la réflexion sera notre clé pour ouvrir toutes ces serrures.

Qu'est-ce qu'un package ?

Sous Delphi 1 et 2, la bibliothèque de composants était déjà une `DLL` modifiée qui contenait la totalité des composants installés dans l'IDE. L'idée était intéressante puisqu'elle permettait, grande nouveauté, de créer des composants avec le même outil de développement et de les intégrer dynamiquement à son espace de travail en une manipulation simple. Il faut rappeler que de nombreux outils de développement n'autorisaient pas, et n'autorisent toujours pas, une telle souplesse et qu'il faut souvent utiliser un autre langage pour étendre leur IDE (lorsque cela est possible). Sous .NET, cette époque semble révolue ; c'est un grand pas en avant.

Toutefois, la stratégie qui fut valable à la sortie de Delphi, quand les composants étaient peu nombreux et les machines sous Win16 peu puissantes, ne pouvait plus être maintenue longtemps : la fameuse « Lib » devenait énorme, il fallait la recompiler à chaque ajout ou suppression d'un composant, et surtout tous les composants étaient présents en même temps, ce qui n'incitait pas à en créer trop, alors même que les bons principes de réutilisation du code imposent le contraire.

Avec Delphi 3, Borland a introduit la notion de *Package* pour offrir cette souplesse indispensable à l'explosion des composants.

Les *packages* ne sont qu'une suite logique de la « Lib » de Delphi 1 et 2, puisque cela revient à avoir plusieurs « Lib » en même temps qu'il est possible de charger ou décharger à volonté sans redémarrer Delphi.

Ainsi, les *packages* sont au départ des `DLL` répondant à un modèle particulier autorisant l'IDE de Delphi à reconnaître les composants ou experts qu'ils contiennent (donc de stocker des définitions de classe). Sous Win32, ces `DLL` étaient compilées dans des fichiers d'extension `.bpl` car ils ne pouvaient pas être considérés comme des `DLL` standards (qui étaient non objet). Sous .NET, les *packages* redeviennent des `DLL` puisque la plate-forme permet de stocker dans tout assemblage les méta-données indispensables à la gestion des classes qui y sont définies et utilisées.

Cette caractéristique leur permet d'être exploités de façon particulière : au lieu de laisser Delphi associer (via son *linker*) le code des composants à l'exécutable, on peut aussi choisir d'utiliser les *packages* à l'exécution (*runtime*), en accompagnement des exécutables qui deviennent alors bien plus petits. Il existe dès lors deux utilisations principales des *packages* : l'une en mode *Runtime* et l'autre en mode *Designtime* (à la conception sous Delphi). Un

troisième mode mixe les deux premiers. Nous allons en voir l'utilité dans les pages qui suivent.

La motivation

Il existe plusieurs raisons objectives d'utiliser les *packages*. En voici les principales.

Réduire la taille du code

Lorsqu'on utilise des *packages runtime*, Delphi n'associe plus le code des composants à celui de l'exécutable. Ce dernier devient donc beaucoup plus petit. En revanche, il faut fournir les packages compilés en accompagnement du logiciel lorsqu'il est déployé. Comme le *Linker* de Delphi est intelligent (c'est un *smart linker*), il supprime le code non appelé et, de fait, un exécutable utilisant les packages de *runtime* semblera plus petit. Néanmoins, l'ensemble des fichiers à fournir sera bien plus gros que le fichier exécutable original sans *package runtime*.

Où se trouve la réduction de code alors ? Elle est toute relative et ne s'apprécie que si vous devez fournir de nombreux exécutables Delphi sur un même site client. À partir d'un moment, plein de petits exécutables partageant les mêmes packages finissent par être bien moins gros que plusieurs exécutables intégrant à chaque fois tout le code nécessaire.

À titre personnel, l'auteur n'a jamais vu de situations réelles où cela soit un vrai avantage. En effet les problèmes de stockage sur disque dur n'existent plus depuis longtemps et la fourniture et le maintien d'un ensemble cohérent de packages présentent un coût important à ne pas négliger.

On peut toutefois voir un avantage dans la séparation du code en unités indépendantes (les *packages*), notamment pour la distribution d'applications via Internet. Dans ce cadre, vous pouvez fournir une fois pour toutes les *packages* de votre application et permettre une mise à jour plus rapide de votre application qui sera de taille plus réduite. La mise à jour d'un package, le cas échéant, sera elle aussi moins consommatrice de bande passante. Toutefois .NET offre aujourd'hui des solutions plus adaptées pour la fourniture de code téléchargeable sur Internet.

Englober un ensemble de composants

Si vous désirez distribuer un ensemble de composants sans fournir le code source, la création d'un package est une excellente solution. D'ailleurs, même si vous souhaitez distribuer le code source, la création de composants passe aujourd'hui systématiquement par celle de packages ! Les composants autonomes « à la Delphi 1 et 2 » n'existent plus, leur installation dans Delphi 3 à BDS passe par leur intégration dans un package, ce que propose l'expert dédié à cet effet dans l'IDE de Delphi. Cependant, nous vous conseillons de bien séparer vos composants en packages différents, afin de ne pas lier la mise à jour d'une classe à celle de toute une bibliothèque énorme.

Si vous créez des composants, alors vous devez tout savoir des packages.

Les différents types de packages

Il existe quatre types de packages, qui peuvent être créés en combinant les options de compilation *ad hoc*. Certains types sont plus utilisés que d'autres comme nous le verrons.

Le tableau 28.1 donne la liste de ces différents types.

Tableau 28.1

Les différents types de packages

TYPE DE *PACKAGE*	UTILITE
Runtime	Ces *packages* contiennent des classes (et des composants). Ils sont nécessaires à l'exécution des applications qui s'en servent et doivent être déployés avec elles. Un tel *package* est marqué par la directive `{$RUNONLY ON}` dans le source du projet.
Designtime	Ces *packages* contiennent des composants, des éditeurs de propriétés, des éditeurs de composants, des experts… Ils sont nécessaires à la conception des applications sous l'IDE de Delphi. Ils ne sont pas exploités à l'exécution des applications et n'ont pas à être distribués avec ces dernières. Ce type de *package* est marqué par la directive `{$DESIGNONLY ON}` dans le source

	du projet.
Designtime et *Runtime*	Ce type de *package* est utilisé lorsqu'il contient du code et des composants qui ne font pas appel à des éléments de conception (éditeurs de propriétés…). Ce type de *package* est le type par défaut créé par l'EDI quand on demande Fichier/Nouveau/Package. Il ne contient aucun marqueur particulier.
Ni *Designtime* ni *Runtime*	Voilà des bêtes bien curieuses ! C'est une espèce excessivement rare de *packages*. Leur unique vocation est d'être exploités par des références directes depuis une application ou depuis l'environnement de conception. De tels *packages* peuvent être intégrés dans d'autres *packages* (bibliothèque de classes par exemple).

D'autres directives sont propres aux *packages* comme `{$DENYPACKAGEUNIT ON}` qui, si elle apparaît dans une unité de code, lui interdit d'être ajoutée à un *package*.

Les fichiers des packages

Les packages sont le fruit de la fusion de différents fichiers. Leur compilation engendre d'autres fichiers et tout cela ne semble pas toujours clair. Fixons alors les choses avec le tableau 28.2, qui liste l'ensemble des fichiers liés directement à la notion de package.

Tableau 28.2

Les fichiers des packages

EXTENSION	TYPE	DESCRIPTION
.DPK	source	***D**elphi **P**acKage* Ce fichier est créé par Delphi lorsque vous appelez l'Éditeur de *Package*. Le DPK est l'équivalent du DPR pour un *package*. Il contient une description des unités plutôt que le code utile lui-même.
.DCPIL	code compilé	***D**elphi **C**ompiled **P**ackage for **I**ntermediate **L**anguage* Ce fichier contient la version compilée d'un *package*. Il stocke les en-têtes et les symboles permettant le bon fonctionnement du *package*. Il contient aussi tous les DCUIL du *package*. Ce fichier est nécessaire pour construire une application qui fait usage du *package* en question.
.DCUIL	unités compilées	***D**elphi **C**ompiled **U**nit for **I**ntermediate **L**anguage* Comme pour un projet classique, les DCUIL contiennent le code compilé de chaque unité du projet.
.DLL	bibliothèque compilée	***D**ynamic **L**ink **L**ibrary* C'est le fichier *runtime* ou *design time* du *package*, équivalent d'une DLL Windows. Si le *package* est de type *runtime*, c'est ce fichier qui doit être distribué avec l'application. Si le *package* est de type *design time*, il faudra fournir ce fichier pour permettre l'utilisation de son code dans l'IDE.

Attention
Pour un *package design time*, si vous ne distribuez pas le code source, vous devrez aussi distribuer le fichier DCPIL avec le fichier DLL.

Installation des packages dans l'IDE

L'installation des *packages* dans l'IDE de Delphi est une chose simple et nécessaire si vous devez intégrer des composants dans votre environnement, que vous en soyez l'auteur ou non.

La première chose à faire consiste à placer les différents fichiers au bon endroit. La table 28.3 liste les emplacements typiques des fichiers principaux d'un package.

Tableau 28.3

Emplacement des principaux fichiers

FICHIER	EMPLACEMENT
`.BPL` *runtime*	Les fichiers *runtime* des *packages* doivent être placés dans un sous-répertoire `\bin` de l'application ou sur un chemin de recherche du système d'exploitation.
`.DLL` *design*	Les fichiers de *packages Design* doivent être placés dans un répertoire commun d'où ils peuvent être facilement gérés. Le mieux reste de créer un sous-répertoire de Delphi lui même et d'y placer toutes les `DLL` de ce type.
`.DCPIL`	Les fichiers des symboles sont placés avec les `BPL` correspondants.
`.DCUIL`	Les unités de code compilé doivent être distribuées si vous distribuez un *package de Design*. Le mieux est de créer un sous-répertoire comme pour les `DLL` de *Design* et de regrouper tous les `DCUIL` de *packages Design* à cet endroit.
`.nfm`	Les éventuelles fiches VCL.NET doivent être distribuées avec les `DCPIL` pour des composants fournis sans code source (avec source, les `nfm` sont forcément présents).

Si vous regardez l'organisation de Delphi lui-même (répertoire `Program Files\Borland\BDS` par défaut), vous remarquerez qu'il existe un répertoire `\Bin` contenant toutes les `DLL` et un répertoire `\Lib` de même niveau contenant tous les `DCPIL` et les `NFM`. Cela donne un bon exemple de répartition de ces fichiers.

Pour l'installation proprement dite des *packages* compilés, Delphi rend les choses très simples, puisqu'il suffit d'ouvrir le dialogue *ad hoc* depuis le menu principal « Composant/Composants .NET Installés », comme le montre la figure 28.1. Vous remarquerez que ce dialogue possède plusieurs onglets ; celui qui nous intéresse ici est « Composants VCL .NET ».

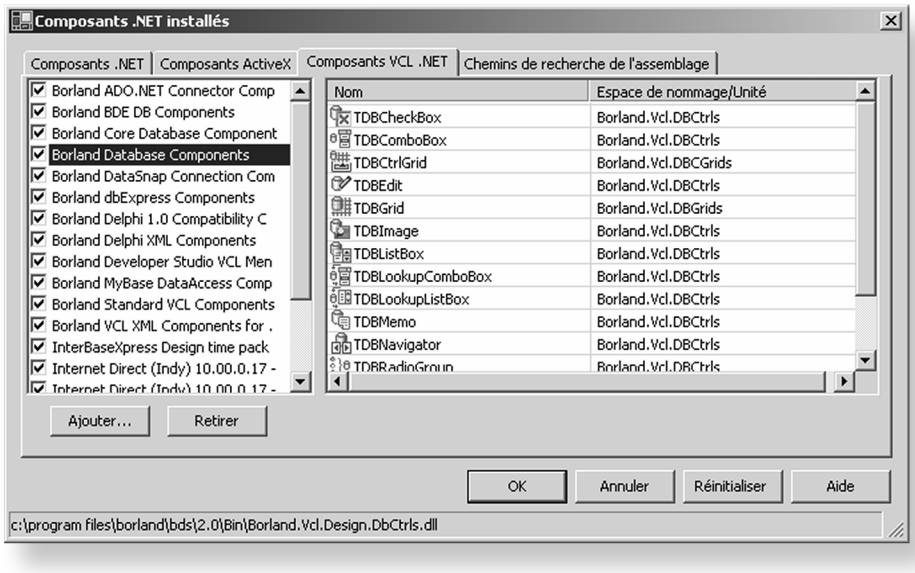

Figure 28.1

Dialogue de recensement des composants

En cliquant sur le bouton Ajouter, on peut choisir un fichier `DLL` package pour l'installer. Si le package contient des composants, ils seront affichés dans la palette d'outils. Le dialogue de la figure 28.1 permet de voir à gauche les packages installés et à droite les éventuels composants qu'ils contiennent, ainsi que l'espace de nom dans lequel ils sont définis. Un même package peut regrouper plusieurs espaces de nom différents. Cela explique pourquoi il y a parfois une différence notable entre le nom de l'espace de nom qu'il faut ajouter à la clause `Uses` d'une application pour utiliser une classe donnée et le nom de la bibliothèque qu'il faut ajouter dans la partie `Références` du projet pour que cet espace de nom soit reconnu par l'application.

Installation alternative depuis l'IDE

Si vous possédez les sources d'un *package*, il est aussi possible de l'installer depuis l'IDE en ouvrant le fichier `DPK` puis en choisissant l'option « Installer un package VCL » depuis le menu local du gestionnaire de projet.

Créer vos propres packages

Avant de créer vos propres packages, vous devez prendre certaines décisions. La première consiste à arrêter le type du package : *runtime*, *design*, aucun des deux. Le choix se fondera sur certains éléments que nous détaillerons plus bas. En second lieu, vous devrez décider comment appeler votre package et où le placer. Enfin, vous devrez décider des unités qui seront intégrées au package ainsi que de l'inclusion d'éventuels autres packages.

L'éditeur de package

Lorsque vous créez un nouveau package ou lorsque que vous modifier un package existant, Delphi ouvre l'éditeur de package, qui sous .NET n'est autre que le gestionnaire de projet (voir figure 28.2).

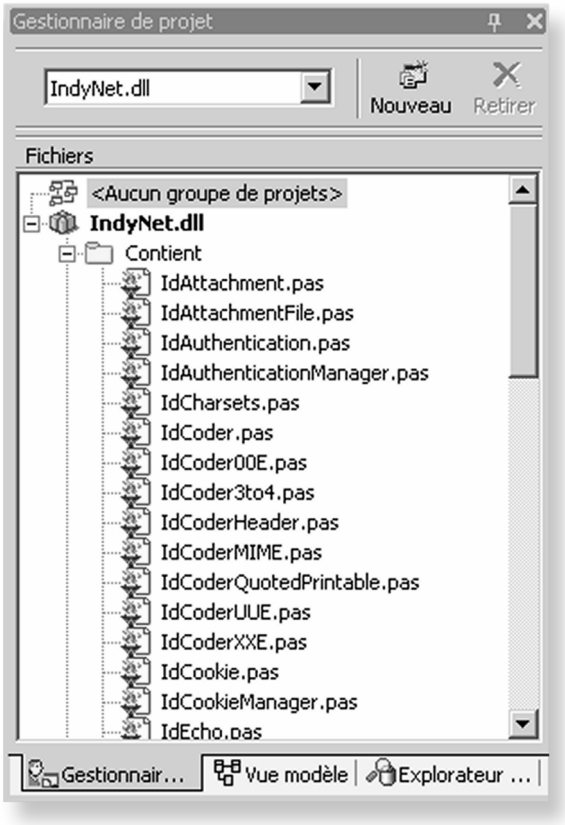

Figure 28.2

Le gestionnaire de projet affichant un package

La majorité du temps, vous créerez un nouveau package en invoquant le menu Fichier/Nouveau puis en sélectionnant Package. Le gestionnaire de projet sera alors affiché, vide bien entendu (à l'exception du package `Borland.Delphi.dll` présent dans la section `Requiert`).

On remarque que l'éditeur contient deux nœuds principaux : `Contient` et `Requiert`.

La section Contient

On spécifie ici l'ensemble des unités de code qui doivent être intégrées dans le package. Quelques règles doivent être respectées :

1. Le package ne doit pas être référencé dans la section `Contient` d'un autre *package* ou dans la clause `Uses` d'une unité de code se trouvant dans un autre package.

2. Les unités listées dans la section `Contient` d'un package, directement ou indirectement (donc dans la clause `Uses` d'une unité), ne peuvent pas être listées dans la section `Requiert` du package. En effet, elles sont déjà liées au package en apparaissant dans la section `Contient`.

3. Vous ne pouvez pas ajouter une unité de code dans la section `Contient` d'un package si elle est déjà intégrée dans la section `Contient` d'un autre *package* utilisé par la même application.

La section Requiert

Vous spécifiez ici tous les packages nécessaires au fonctionnement du vôtre et qui n'apparaissent pas dans la section `Contient`. On peut voir cette section comme l'équivalent de la clause `Uses` dans une unité de code.

En général, et au minimum, les packages que vous créerez intégreront la bibliothèque `Borland.Delphi.dll`. En effet, c'est ce package qui contient la base du code *runtime* de Delphi, notamment l'espace de nom `Borland.Delphi.System` qui déclare de nombreux éléments indispensables à toute application Delphi .NET. Ce fichier est requis et non contenu. Les fichiers contenus sont stockés dans le package compilé, les fichiers requis doivent être déployés avec le package.

La stratégie de conception des packages

Comme indiqué plus haut dans cet article, la première chose à savoir quand on crée un package est le type de celui-ci. Nous allons voir ici comment faire ce choix.

Les packages de composants Design time et Runtime

C'est le scénario classique pour celui qui conçoit des composants car :

- Il souhaite que les programmeurs qui utiliseront ses composants puissent compiler et lier le code de ceux-ci à leurs propres applications.

- Il souhaite qu'ils puissent aussi référencer ses composants sans les lier et en fournissant aux utilisateurs le *package runtime*.

- Il ne souhaite pas que les utilisateurs de son package soient obligés de « traîner » le code de *design* (éditeurs, experts…) dans leurs applications finales, ce qui serait inutile.

- Il veut aussi que le code de ses composants soit compatible avec la philosophie imposée depuis Delphi 6 et qui réclame que le code de *Design* soit séparé du code de *Runtime* et ce afin de créer des composants éventuellement portables vers Delphi 7 ou Delphi .NET.

Ici, le développeur n'a guère le choix de la structure à choisir. Il faudra créer deux packages distincts, l'un pour le *Design* et l'autre pour le *Runtime*.

Le dessin (figure 28.3) schématise le jeu d'inclusion entre les deux types de packages.

Figure 28.3

Organisation des packages Runtime et Design time

Comme le montre la figure, le package de *Design* `MonPackageLib80.Dpk` intègre à la fois les fonctions de *Design* proprement dites (éditeurs divers) et l'enregistrement des composants (appel à `RegisterComponents` dans l'unité `MonCompReg.pas`), ainsi qu'une référence au package *Runtime* `MonPackStd80.Dpk`. Ce dernier contient le code des composants eux-mêmes (`Composant1.pas` et `Composant2.pas` dans notre exemple), ainsi que des unités de services symbolisées ici par l'unité `Mesfonctions.pas`. Vous noterez que dans une telle organisation, nous créons une unité spéciale pour l'enregistrement des composants et que, de fait, la procédure `Register` ne doit plus être située dans les unités des composants eux-mêmes.

Le package de *Runtime* est donc intégré dans le package de *Design* par une référence dans la section `Requiert` de ce dernier. Pour que ce montage fonctionne, vous devez modifier correctement les options de chacun des deux packages.

Pour ce faire, vous devez ouvrir le code source du projet et saisir dans la liste des options soit `{$DESIGNONLY ON}` pour le package de *Design*, soit `{$RUNONLY ON}` pour le package de *Runtime*. Sous Delphi 7, il existe un paramétrage par le dialogue des options du projet pour effectuer cette manipulation. Sous Delphi .NET, il faut intervenir dans le source directement (cela évoluera certainement au fil des version de l'IDE).

L'enregistrement des composants

Lorsque vous créez un nouveau composant, Delphi ajoute automatiquement une fonction `Register()` qui fait un appel à `RegisterComponents()`.C'est cette procédure qui permet à l'IDE de repérer un code à installer. Une fois qu'un package a été créé et est ouvert dans l'IDE, vous pouvez ajouter un nouveau composant par le dialogue Fichier/Nouveau/Autre, choisir le dossier « Projets Delphi pour .NET » puis le sous-dossier « Nouveaux Fichiers » et enfin, dans la partie droite du dialogue, sélectionner « Composant VCL » comme le montre la figure 28.4.

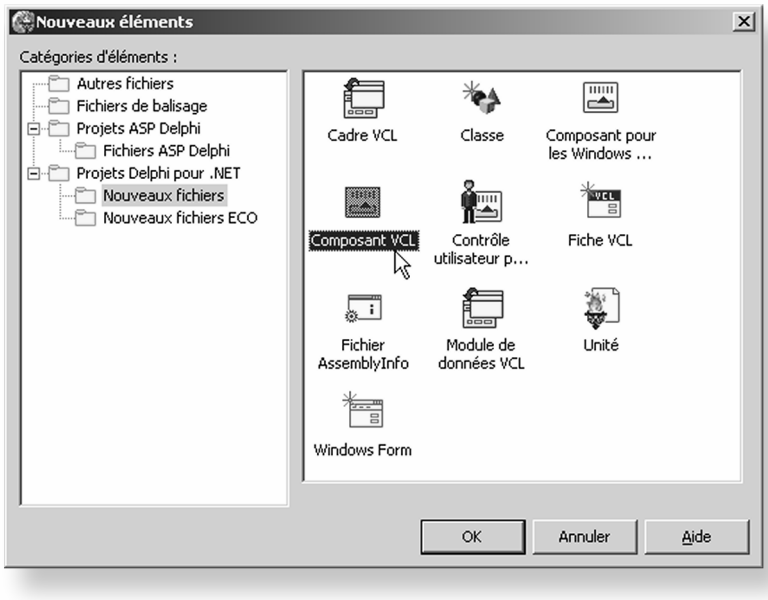

Figure 28.4

Création d'une unité de composant VCL

Lors de la création d'un nouveau composant, Delphi propose un dialogue permettant de saisir son nom, l'emplacement de son unité et la classe dont il descend (voir figure 28.5). Une fois ces informations saisies, un fichier est ouvert dans l'éditeur de code source.

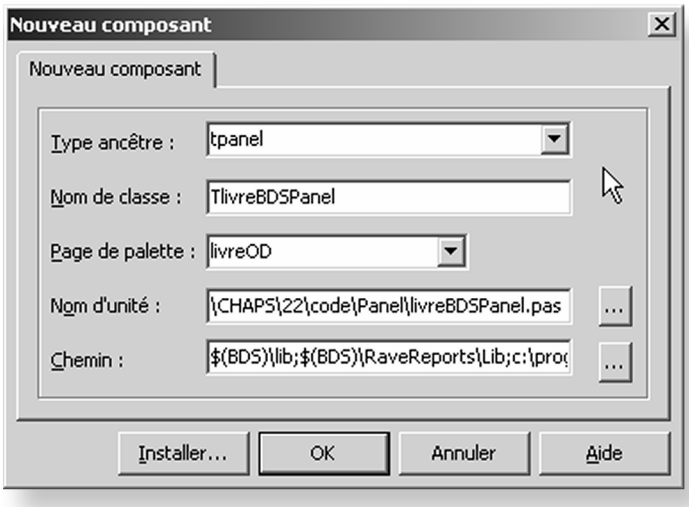

Figure 28.5

La création d'un nouveau composant

Dans le montage qui vient d'être présenté, cette stratégie ne s'applique plus. Il convient de déplacer le code de `Register()` dans une unité séparée, qui recensera l'ensemble des composants s'il y en a plusieurs, ainsi que l'enregistrement des éditeurs et experts éventuels.

Cette unité, comme nous l'avons vu plus haut, apparaît dans la section `Contient` du package de *Design*.

En faisant de la sorte, le code d'enregistrement se trouve bien là où il doit être : dans le package de *Design*. Il ne sera pas intégré dans les applications qui utilisent vos composants, et mieux encore, cela interdira d'installer vos composants dans Delphi à partir du fichier `BPL` si celui-ci est distribué avec les applications (au lieu d'être lié à l'exécutable), ce qui constitue une protection supplémentaire pour gérer vos licences.

Remarque
Certains développeurs rusés pourraient penser qu'il suffit de recréer une unité d'enregistrement puis un *package* de *Design* intégrant votre `BPL` pour contourner la chose... En effet, avec une telle astuce, il sera possible de monter les composants dans la palette de Delphi et même de jouer un peu avec, mais il sera impossible de compiler une application car il manquerait les `DCUIL` et le `DCPIL`...

Les directives de compilation spécifiques aux packages

Certaines directives sont spécifiques aux packages, elles sont incluses soit dans le code même du package, soit dans les unités qui peuvent être utilisées par un package. La liste de ces directives est données au tableau 28.4.

Tableau 28.4

Directives spécifiques aux packages

DIRECTIVE	LOCALISATION	DESCRIPTION
`{$G-}` ou `{$IMPORTEDDATA OFF}`	Unité contenue dans un *package*	La directive `{$G-}` désactive la création de références aux données importées. `{$G-}` augmente les performances d'accès à la mémoire, mais empêche l'unité paquet dans laquelle elle apparaît de faire référence à des variables se trouvant dans d'autres paquets.
`{$DENYPACKAGEUNIT ON}`	Unité contenue dans un *package*	La directive `{$DENYPACKAGEUNIT ON}` empêche l'unité dans laquelle elle apparaît d'être placée dans un *package*.
`{$WEAKPACKAGEUNIT ON}`	Unité contenue dans un package	Force le code compilé de l'unité à être intégré dans l'exécutable de façon statique plutôt que de rester dans le *package* (il est placé dans le `DCPIL` plutôt que dans le `BPL`). Cette directive a été créée pour régler des problèmes d'accès à des `DLL` éventuellement non existantes sur la machine au sein du noyau de la VCL. On ne s'en sert donc que de façon rarissime.
`{$DESIGNONLY ON}`	Source du projet (`DPK`)	Directive de *package*. Compile en mode *Design* seulement.
`{$RUNONLY ON}`	Source du projet (`DPK`)	Directive de *Package*. Compile en mode *Runtime* seulement.
`{$IMPLICITBUILD OFF}`	Source du projet (`DPK`)	Empêche le *package* d'être reconstruit implicitement par Delphi lors d'une reconstruction. Utilisez cette directive dans les fichiers `DPK` lors de la compilation de paquets qui fournissent des fonctionnalités de bas niveau et qui ne sont pas modifiées fréquemment ou dont le source n'est pas distribué.
`{$DESCRIPTION 'texte'}`	Source du projet (`DPK`)	Permet de donner une description textuelle à un paquet. Cette ligne sera affichée dans la fenêtre de gestion des packages de l'EDI. Nota : la chaîne doit faire moins de 256 caractères.

Conventions de nom

Les packages vont souvent par paire, ne serait-ce que pour les composants, comme nous l'avons vu plus haut. De plus, les packages sont dépendants de la version de Delphi sous Win32,

c'est-à-dire qu'un package compilé avec Delphi 7 ne fonctionnera qu'avec des applications compilées sous Delphi 7. Il peut donc être important de respecter une stratégie de nom lorsque vous écrivez du code portable Win32/.NET

Il est dès lors essentiel de pouvoir faire la différence entre les packages compilés pour des versions différentes de Delphi. Il n'y aucune règle fixe, mais il existe quelques conventions relativement bien suivies.

La première consiste à placer la version de Delphi sur deux chiffres dans le nom des packages. Par exemple, le noyau de la VCL est contenu dans `VCL70.bpl` pour Delphi 7. Pour une raison mystérieuse, Delphi 6 échappait à cette belle logique. Néanmoins, si les développeurs de Borland nous montrent souvent des voies intéressantes, ne tombons pas dans le piège de la « gurutisation » et ne les suivons pas lorsqu'ils manquent à leurs propres règles. Si votre package n'est voué qu'à .NET, alors optez pour un nom assez long, clair, contenant l'espace de nom. Sous Delphi .NET, les packages s'appellent par exemple : `Borland.Vcl.Buttons`, `Borland.Data.Common`... Nommez vos packages avec le nom de votre société ou le vôtre, par exemple `E-Naxos.MKQB.DataDrivers`.

La seconde convention consiste à bien identifier les packages *Design time* et *Runtime*.

L'une des normes suivies est celle qui consiste à ajouter, avant le numéro de version de Delphi, un sigle de trois lettres : `Lib` pour les packages de *Design* et `Std` pour les packages *Runtime*.

On trouve aussi une autre norme, qui consiste à utiliser `Dcl` pour les packages de *Design* et soit rien soit les lettres `Ctl` pour le package *Runtime*.

Enfin, il est habituel de placer un préfixe de trois lettres indiquant le nom du créateur, mais avec les espaces de nom de .NET, le nom complet de l'auteur (ou de sa société) font encore mieux l'affaire. Reste à ajouter au milieu de tout çà le nom du package lui-même.

De fait, les deux packages d'un produit de comptabilité fictif TRUC créé par le célèbre John Smith pour Delphi 7 s'appelleraient :

- `JSTrucLib70.dpk` pour le *package* de *Design*,
- `JSTrucStd70.dpk` pour le *package Runtime*.

Pour Delphi .NET, ces noms seraient plus parlants et pourraient être :

- `JSmith.Accounting.Truc.80.Lib.dpk`,
- `JSmith.Accounting.Truc.80.Std.dpk`.

Vous pouvez adopter d'autres conventions, tant qu'elles sont claires et permettent de bien gérer la version de Delphi et la finalité (*Design* ou *Runtime*). Loin des acronymes difficiles à comprendre, l'auteur vous conseille d'être clair jusqu'au bout et d'utiliser des suffixes longs et explicites tels que *Design* et *Runtime* : pourquoi faire compliqué et sibyllin quand on peut faire simple et parlant ?

Les packages dynamiques

Déjà sous les versions Win32 de Delphi, il était possible d'exploiter les packages dynamiquement, c'est-à-dire comme on le faisait d'une `DLL`. La différence venait seulement de la possibilité de stocker et donc de pouvoir exploiter des classes au lieu de simples procédures et fonctions. Sous .NET, nous l'avons vu ici, les packages Delphi redeviennent des `DLL` puisque la plate-forme est désormais objet. De fait, il est toujours possible, et presque plus qu'avant, d'utiliser les *packages* dynamiquement dans les applications.

Qu'est-ce que le chargement dynamique de *packages* (ou de `DLL`) ? Il s'agit de la possibilité de charger une bibliothèque lors de l'exécution de l'application, sans que cette dernière ne soit liée au package lors de sa propre compilation. Le principe est utilisé pour les `DLL` depuis longtemps ; c'est même leur fonction première puisque `DLL` signifie *Dynamically Linked Library*, soit bibliothèque [de code] liée [à l'application] dynamiquement. Si cela se limitait aux ressources et aux fonctions sous Win16 et Win32, cela s'étend aux désormais aux classes sous .NET.

À quoi cela sert-il ? Simplement à étendre les fonctions d'un logiciel sans le recompiler, mais aussi à l'ouvrir à des parties tierces. Tout le monde connaît les filtres graphiques de Photoshop d'Adope ou Paint Shop Pro de Jasc Software ou bien les extensions VST du logiciel musical Cubase de Steinberg par exemple. Les exemples sont innombrables et tous les « grands » logiciels proposent de telles extensions appelées aussi *plugins* (ou *plug'in*), ce qu'on pourrait traduire approximativement par « branchez-le dedans ». Ces extensions représentent un double avantage énorme pour le concepteur d'une application : son logiciel est « ouvert », perfectible, extensible (des mots qui sonnent bien sur une brochure commerciale) et avec un peu de chance et de popularité, il laisse à d'autres concepteur le soin de développer des modules qui rendront son logiciel encore plus attractif et populaire… Futés les *plugins* ?

Oui, et à tous les niveaux, du marketing à la technique. Pour une fois qu'une solution de technicien peut plaire aux commerciaux, pourquoi s'en priver !

Les divers types de plugins

Selon ce qu'on désire faire, il existe plusieurs sortes de *plugins*. On peut au moins les classer en deux grandes catégories. La première est celle des *plugins* contenant un (ou plusieurs) descendant(s) de classes précises devant correspondre à un modèle, un moule spécifié. Ce sont les plus classiques. On trouve dans cette catégorie les fameux filtres Photoshop et les effets spéciaux ou les simulations de synthétiseurs VST de Cubase dont nous parlions plus haut. Dans ce cadre précis, le concepteur du logiciel a créé une enveloppe pour un certain type de fonctions bien défini. Le développeur de *plugin* doit reprendre à l'identique le même modèle, les mêmes noms de procédures et fonctions. La plupart des logiciels Win32 qui utilisent ce principe le font avec des `DLL` classiques, donc non objet, mais cela ne change rien au principe. Sous .NET, on peut regrouper les fonctions dans une classe abstraite servant de moule de fabrication. Dans ce type de *plugin*, le plus répandu, seules certaines fonctions du logiciel de base sont accessibles et le *plugin* lui-même est limité à une fonction précise. Un filtre graphique Photoshop ne peut pas ajouter une fonction d'importation `GIF` par exemple, il ne peut que travailler sur le *bitmap* que lui passe Photoshop, le transformer puis le retourner modifié.

La seconde catégorie de *plugin* est plus rare, certainement car elle prend tout son sens avec les objets et que les *plugins* objets vont réellement prendre leur essor avec .NET. Ici, il n'y a plus de fonction précise attribuée au *plugin* ; il peut faire à peu près tout ce qu'il veut. C'est le modèle adopté par Delphi depuis la version 3 sous Win32 avec la naissance des packages : Delphi a été ainsi conçu que vous pouvez ajouter des composants à l'EDI dont personne chez Borland ne sait rien et les utiliser comme s'ils en faisaient partie depuis toujours. Pourtant, un descendant de `TDataset` n'a aucun rapport avec un `TShape`, qui lui-même n'a aucun point commun avec la classe `TTimer`. Pourtant, tout cela peut être utilisé, manipulé et paramétré naturellement sous l'EDI. Les propriétés sont affichables dans l'inspecteur d'objets, les gestionnaires d'événements sont créés en respectant les déclarations du code source du composant. Quelle magie permet tant de souplesse ? Elle tient en un sigle : les RTTI ou *Run Time Type Information* (information de type à l'exécution), système autorisant une application Delphi à se connaître elle-même, à accéder aux déclarations de son propre code à l'exécution, alors

qu'elle est compilée et ne contient aucune copie cachée du code source original.

Les RTTI prennent tout leur sens avec les objets et Delphi tire profit de ce mécanisme depuis des années. Avec .NET, les RTTI « à la Borland » tirent leur révérence et laissent la place à la Réflexion qui en reprend le principe. La différence est malgré tout de taille puisque, autant les RTTI étaient spécifiques à Delphi faute d'un tel mécanisme sous Win32, autant la Réflexion est un procédé totalement intégré au framework .NET et fonctionnant avec tous les langages de la plate-forme.

Les fonctions de Réflexion étant désormais intégrées à la plate-forme, le système des RTTI typiquement Borland n'existe plus. Les fonctions spécifiques à la gestion des packages ont aussi disparu de l'unité `SysUtils` puisque celles destinées à la gestion des `DLL` et au mécanisme de Réflexion permettent de tout faire. De plus, les unités comme SysUtils sont spécifiques à la VCL Win32 et la VCL.NET alors que les packages peuvent être utilisés aujourd'hui sous BDS pour créer des composants Windows Forms ou créer des librairies de classes utilisables depuis C# ou d'autres langages de la plate-forme. De fait, les packages, en redevenant des DLL standard, prennent une nouvelle épaisseur, une universalité qui, il faut le dire, leur faisait cruellement défaut sous Win32.

Tout faire… oui, mais certainement plus comme avant… Et c'est là que les choses se corsent, comme d'habitude. La Réflexion faisant partie de .NET, les documentations Borland n'en font que très peu état, et les sites parlant de la Réflexion de .NET ne font que très peu état de Delphi… Le développeur doit ici faire le lien entre des exemples C# commentés en anglais et Delphi, en tentant d'adapter au mieux ce qu'il aura pu comprendre. Rien n'est impossible, c'est ce qu'a fait l'auteur pour vous en parler ici, vous pourriez le faire vous-mêmes certainement. Toutefois, il est tellement plus pratique de n'avoir qu'à ouvrir un livre où, en français, les explications seront données directement pour Delphi… C'est ce que nous allons voir maintenant au travers d'un exemple de code complet.

L'application exemple

Discuter doctement est parfois intéressant, mais voir concrètement « comment ça marche » a toujours été primordial. C'est pourquoi nous allons ici détailler un exemple qui permettra d'introduire les concepts sous-jacents.

L'application exemple est en fait un groupe de trois modules distincts. Nous y trouverons :

- le code d'un composant et son package ;

- le code de l'application permettant de tester le composant ;
- le code de l'application chargeant dynamiquement le package contenant le composant et l'utilisant.

Le composant

Rien ne sert de faire compliqué. Il nous faut une classe contenue dans un package pour mettre en évidence les mécanismes de réflexion et l'utilisation des packages dynamiques. Toutefois, une classe visuelle sera plus « photogénique » pour les captures d'écran de cet ouvrage, ce qui nous a amené à développer un composant dérivé de `Tpanel` : un `Panel` qui donne l'heure et la date et qui sait se redimensionner tout seul en fonction de la taille de la fonte et du format choisi pour l'affichage.

Nous commençons par créer un nouveau projet package par le menu Fichier/Nouveau/Package et appelons cette `DLL` package `LivreOD_Panel`. Nous ajoutons ensuite un composant au *package* par le menu Fichier/Nouveau/Autre puis, dans le dialogue qui s'ouvre, nous faisons une sélection identique à celle montrée figure 28.4 (Création d'une unité de composant VCL). Nous nommons cette unité `LivreBDSPanel`, après avoir indiqué notre souhait de créer un descendant de `TPanel`, que nous nommons `TLivreBDSPanel`.

Le fonctionnement de ce composant est simple, mais démontre de nombreux aspects propres au développement de ce type de classe particulier. Étudions son code source.

L'interface

L'interface de notre composant est formé par l'ensemble des déclarations débutant au mot-clé `Interface` et s'arrêtant juste avant le mot-clé `Implementation`. La seule chose qui précède ces déclarations est la clause `Unit` suivie du nom de l'unité. Le code ci-dessous contient les commentaires nécessaires à sa compréhension.

```
unit livreBDSPanel; // nom de l'unité, définit aussi son espace de nom

interface

// Liste des espaces de nom utilisés par le composant
uses
  SysUtils, Classes, System.ComponentModel, Borland.Vcl.Controls,
  Borland.Vcl.ExtCtrls, Borland.Vcl.Dialogs ;

// Déclaration de la classe du composant
```

```
type
  TlivreBDSPanel = class(TPanel)
  private
    // Déclaration des champs privés
    ftimer : TTimer;          // un timer pour afficher l'heure à intervalles réguliers.
    fStampFormat : string;    // chaîne de format date et heure
    fAutoSize : boolean;      // mode retaillage automatique
    procedure SetStampFormat(value:string); // Setter du format date heure
    procedure SetInterval(value:integer);     // Setter de l'intervalle
    function GetInterval:integer;                  //Getter de l'intervalle
    procedure InternalOnTimer(sender:tobject); // Event du timer
  protected
    // declarations protégées
    Procedure CreateWnd; override; // redefinition de CreateWnd
    function AutoWidth : integer;   // Retourne la largeur du format
    function AutoHeight : integer; // Retourne la hauteur du format
    Procedure FontChanged(sender:Tobject); // Si la fonte change…
  public
    // Déclarations publiques
    constructor create(aowner:tcomponent); override; // constructeur
    destructor destroy; override; // destrcuteur
  published
    // Propriétés publiées
    procedure AboutBox; // Boîte "A propos"
    property StampFormat : string   // Format d affichage date et heure
              read fStampFormat write SetStampFormat;
    property Interval : integer     // Intervalle du timer
              read GetInterval write SetInterval default 1000;
    property Font;    // fonte
    property Height; //Hauteur
    property AutoSize : boolean // mode taille automatique
              read fAutoSize write fAutosize default true;
  end;

 procedure Register; // n'est plus utile dans ce contexte mais permet
                     // d'installer le composant sous EDI si nécessaire
```

L'implémentation

L'implémentation est la partie qui débute au mot-clé `Implementation` et se termine au `end` final de l'unité. Elle comporte tout le code fonctionnel des classes définies dans la partie `Interface`. Suivant le même principe que précédemment, nous avons placé les commentaires dans le code qui suit.

```
implementation

// procedure d'enregistrement du composant dans l'IDE
procedure Register;
begin
  RegisterComponents('livreOD', [TlivreBDSPanel]);
end;
```

```
{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.create
 Arguments: aowner:tcomponent
 Result:    None
 Usage:     Constructeur de la classe, initialisation des champs
-------------------------------------------------------------------------------}
Constructor TlivreBDSPanel.create(aowner:tcomponent);
begin
 Inherited create(aowner);
 ControlStyle := ControlStyle - [csAcceptsControls,csSetCaption];
 fStampFormat := LongDateFormat + ' '+ LongTimeFormat;
 Alignment := tacenter;
 fAutoSize := true;
 font.OnChange := FontChanged;
 ftimer:=TTimer.create(self);
 ftimer.Interval := 1000;
 ftimer.OnTimer := InternalOnTimer;
 ftimer.Enabled := true;
 InternalOnTimer(nil);
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.CreateWnd
 Arguments: None
 Result:    None
 Usage:     Modifier les propriétés visuelles une fois le parent fixé
-------------------------------------------------------------------------------}
Procedure TlivreBDSPanel.CreateWnd;
begin
 inherited CreateWnd;
 if fAutoSize then begin
  Width := AutoWidth;
  Height := AutoHeight;
  end;
end;

{-------------------------------------------------------------------------------
 Procedure: tLivreD7TimePanel.destroy
 Arguments: None
 Result:    None
 Usage:     Destructeur
-------------------------------------------------------------------------------}
Destructor TlivreBDSPanel.destroy;
begin
 ftimer.Enabled:=false;
 ftimer.Free;
 Inherited destroy;
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.InternalOnTimer
 Arguments: sender:tobject
 Result:    None
 Usage:     OnTimer interne, mise à jour affichage
```

```
-------------------------------------------------------------------------------}
procedure TlivreBDSPanel.InternalOnTimer(sender:tobject);
begin
 caption := FormatDateTime(fStampFormat,now);
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.SetInterval
  Arguments: value:integer
 Result:    None
 Usage:     setter de la propriété Interval
-------------------------------------------------------------------------------}
procedure TlivreBDSPanel.SetInterval(value:integer);
begin
 if Interval<>value then
  begin
   ftimer.Enabled:=false;
   ftimer.Interval := value;
   ftimer.Enabled:=true;
  end;
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.GetInterval
 Arguments: None
 Result:    integer
 Usage:     getter de la propriété Interval
-------------------------------------------------------------------------------}
function TlivreBDSPanel.GetInterval:integer;
begin
 Result := ftimer.Interval;
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.SetStampFormat
 Arguments: value:string
 Result:    None
 Usage:     setter de la propriété StampFormat
-------------------------------------------------------------------------------}
procedure TlivreBDSPanel.SetStampFormat(value:string);
begin
 if value<>fStampFormat then
  begin
   ftimer.Enabled:=false;
   fStampFormat:=value;
   InternalOnTimer(nil);
   ftimer.Enabled:=true;
  end;
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.AboutBox
 Arguments: None
 Result:    None
 Usage:     Pour démonstration de l'appel d'une méthode
```

```
-------------------------------------------------------------------------------}
procedure TlivreBDSPanel.AboutBox;
begin
Showmessage('Composant de test'#13#10'+
     Livre Delphi .NET'#13#10'+
     Chargement dynamique de package'#13#10'(c) 2004 OD');
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.AutoWidth
 Arguments: None
 Result:    None
 Usage:     Calcule la largeur du contrôle automatiquement
-------------------------------------------------------------------------------}
function TlivreBDSPanel.AutoWidth: integer;
begin
 canvas.Font.Assign(font);
 result := canvas.TextWidth(FormatDateTime(fStampFormat,now))+10;
end;

{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.AutoHeight
 Arguments: None
 Result:    None
 Usage:     Calcule la hauteur du contrôle automatiquement
-------------------------------------------------------------------------------}
function TlivreBDSPanel.AutoHeight: integer;
begin
 canvas.Font.Assign(font);
 result := canvas.TextHeight(FormatDateTime(fStampFormat,now))+4;
end;



{-------------------------------------------------------------------------------
 Procedure: TlivreBDSPanel.FontChanged
 Arguments: None
 Result:    None
 Usage:     Nous ajoutons ce code pour enregistrer la classe
            automatiquement lorsque le package appelant sera chargé
-------------------------------------------------------------------------------}
procedure TlivreBDSPanel.FontChanged(sender: Tobject);
begin
 if fAutoSize then begin
  Width := AutoWidth;
  Height := AutoHeight;
  end;
end;

// Initialisation de l'unité
// Utile uniquement si installation du composant dans l'IDE
Initialization
 RegisterClass(TlivreBDSPanel);

// Finalisation de l'unité
```

```
Finalization
 UnRegisterClass(TlivreBDSPanel);
end.
```

L'application de test du composant

Lorsqu'on développe un composant, il est indispensable de développer conjointement une application hôte de test. La raison en est facilement compréhensible : il est bien plus aisé de voir et corriger le comportement du composant au travers de l'application de test que de l'installer dans l'IDE, se rendre compte des problèmes, le désinstaller, le modifier, le réinstaller…

Une telle application de test consiste à référencer l'unité du composant dans le code d'une fenêtre `TForm`, puis à ajouter sur celle-ci boutons, champs d'édition et autres, nécessaires à la modification et à l'accès à toutes les propriétés et méthodes. Pour cet exemple, nous nous sommes contentés de disposer deux boutons, l'un pour créer l'instance, l'autre pour la détruire, et d'ajouter un moyen de changer la taille de la police pour voir en action l'effet du mode de retaille automatique.

Visuellement, l'application de test est telle que le montre la figure 28.6. Le code n'est guère passionnant et nous vous laissons l'étudier seul puisque l'application est fournie sur le CD-Rom du livre.

Figure 28.6

L'application de test du composant

L'application hôte du plugin

Une fois le package du composant compilé, nous disposons d'une `DLL` qui pourrait être installée dans l'IDE pour ajouter le composant à la palette d'outils de Delphi. Cependant, nous allons

ici nous en servir d'une façon moins conventionnelle, en tant que *plugin*.

Il s'agira ici d'une gestion de *plugin* de la seconde catégorie évoquée plus haut, c'est-à-dire d'un mécanisme dans lequel l'application hôte ne sait rien de la classe qui sera créée et ne dispose d'aucune convention de dialogue avec celle-ci en dehors des propriétés et méthodes de la classe mère dont elle hérite. Le composant *plugin* pourrait dessiner un mouton ou jouer le `MP3` de la Traviata chantée en javanais, cela n'aurait aucune importance à la différence des *plugins* de la première catégorie qui doivent avoir une fonction précise. En ce sens, notre application hôte va se comporter exactement comme l'IDE de Delphi, qui peut charger dynamiquement des *packages* de composants et les afficher dans sa palette d'outils. L'IDE ne sait rien de ce qu'ils font ; il sait juste, convention minimale et minimaliste, qu'ils sont dérivés de `TComponent`.

La figure 28.7 montre l'application hôte en fonctionnement une fois le package chargé et l'instance du composant créée.

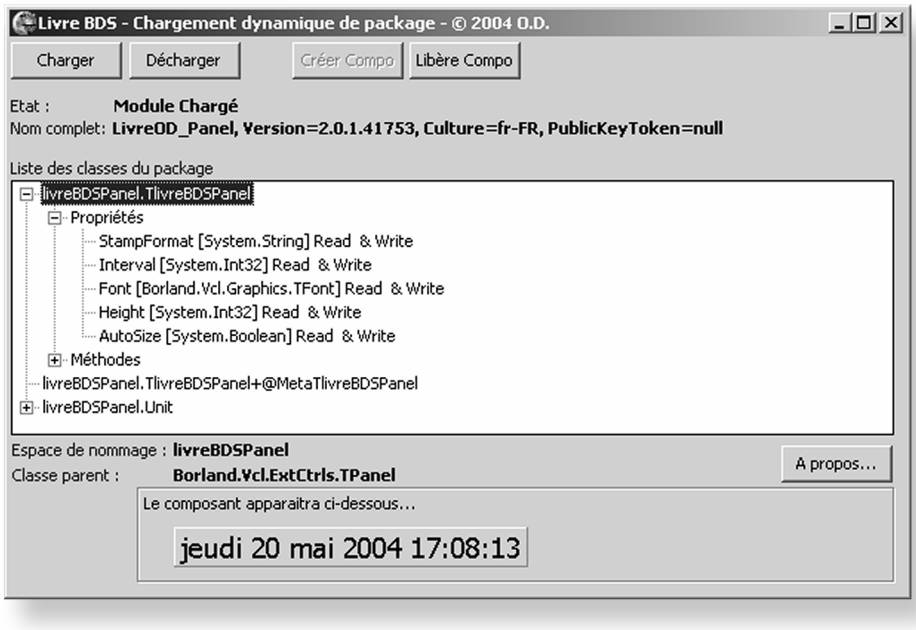

Figure 28.7

L'application hôte du plugin

Quels sont les principes utilisés dans l'application hôte ? Principalement celui de la Réflexion, le reste n'étant que mise en page.

Voici les composants qui constituent la fiche de l'application telle que vous pouvez la voir sur la figure 28.7 et le code correspondant.

Bouton Charger

Sous ce bouton se cache une bonne partie du code de Réflexion de l'exemple, car après avoir chargé la `DLL`, le `treeview` est construit en analysant les classes contenues dans celle-ci.

Voici le code que nous allons détailler plus bas :

```
procedure TForm1.btnLoadPackageClick(Sender: TObject);
var tp : array of System.Type;
    mi : array of System.Reflection.MethodInfo;
    pi : array of System.Reflection.PropertyInfo;
    t,tt,ttt  : ttreenode;
    i,ii  : integer;
    s  : string;
begin
 if not assigned(fPlugin) then begin // si le plugin n'est pas encore chargé
  fPlugin := Assembly.LoadFrom(kPackageName); // chargement
  lPackDesc.Caption := fPlugin.GetName.ToString;
  tp := fPlugin.GetTypes;
  for i := low(tp) to high(tp) do begin
      // noeud classe
      t:=tvClasses.Items.AddObject(nil,tp[i].FullName,tp[i]);
      // Propriétés dynamiques, publiques et déclarées dans le type (non héritées)
      pi:=tp[i].GetProperties(BindingFlags.DeclaredOnly
          or BindingFlags.Instance or BindingFlags.Public);
      if assigned(pi) and (high(pi)>low(pi)) then begin
       tt:=tvClasses.Items.AddChild(t,'Propriétés');
       for ii := low(pi) to high(pi) do begin
        if pi[ii].CanRead then s:='Read ' else s:='';
        if pi[ii].CanWrite then s:=s+' & Write';
        ttt:=tvClasses.Items.AddChild(tt,pi[ii].Name+
                ' ['+pi[ii].PropertyType.tostring+'] '+s);
       end;
       end;
      // Méthodes publiques déclarées dans le type (non héritées)
      mi:=tp[i].GetMethods(BindingFlags.DeclaredOnly
           or BindingFlags.Instance or BindingFlags.Public
           or BindingFlags.Static);
      if assigned(mi) and (high(mi)>low(mi)) then begin
       tt:=tvClasses.Items.AddChild(t,'Méthodes');
       tt.ImageIndex:=-1;
       for ii := low(mi) to high(mi) do begin
        ttt:=tvClasses.Items.AddChild(tt,mi[ii].Name+
             ' ['+mi[ii].tostring+']');
       end;
       end;
      end;
  end;
 UpdateState; // mise à jour de l'affichage
end;
```

Pour utiliser la Réflexion dans une application, il faut ajouter l'espace de nom `System.Reflection` dans la clause `Uses` de celle-ci, ce que nous avons fait et qui n'est pas visible ici.

Pour charger la `DLL` package, nous utilisons la méthode de classe `Assembly.LoadAssemblage(<nom>)`, retournant une instance de type `Assembly` que nous stockons dans une variable de même type appelée ici `fPlugin`. Ce chargement ne s'effectue que si cette variable est non initialisée (si le package n'est pas déjà chargé), ce qui explique le code :

```
if not assigned(fPlugin) then begin // si le plugin n'est pas encore chargé
  fPlugin := Assembly.LoadFrom(kPackageName); // chargement
```

La constante `kPackageName` est définie dans l'application car ici nous nous limiterons au chargement de notre package exemple. Il faudrait peu de code en réalité pour que l'application hôte puisse charger n'importe quelle `DLL` .NET (en offrant par exemple une boîte de dialogue d'ouverture de fichier).

Une fois l'assemblage chargé, nous allons construire le `treeview` en plaçant en nœuds fils de la racine toutes les classes contenues dans celui-ci. Sous chaque classe, nous créerons un sous-nœud fils pour les propriétés et un autre de même niveau pour les méthodes.

La première chose à obtenir est l'ensemble des informations des classes définies dans le package. Cela est effectué par le code suivant :

```
  tp := fPlugin.GetTypes;
  for i := low(tp) to high(tp) do begin
```

La variable `tp` a été déclarée comme suit :

```
var tp : array of System.Type;
```

Il s'agit d'un tableau dynamique dont les éléments sont des variables de type `System.Type`. Pour boucler sur l'ensemble des éléments du tableau, Delphi ne propose pas d'itérateur `ForEach`, mais il est facile de créer une boucle sur un tableau dynamique à l'aide de `Low` et `High` comme le montre le code ci-dessus.

C'est à l'intérieur de cette boucle que va s'inscrire le reste du code de la méthode. Pour matérialiser le chargement des informations de chaque classe, nous créons un nœud dans le `treeview`, fils d'une racine nulle (`nil`). Dans ce nœud, nous plaçons à la fois le nom complet de la classe (préfixé par son espace de nom) et l'objet `Type` lui-même, ce qui permettra de retrouver facilement les informations de la classe cliquée dans le `treeview` :

```
    // noeud classe
    t:=tvClasses.Items.AddObject(nil,tp[i].FullName,tp[i]);
```

La variable `t` est de type `TTreeNode` et permettra de créer les nœuds enfants pour les propriétés et méthodes.

Il faut maintenant charger la liste des propriétés de la classe en cours d'inspection :

```
// Propriétés dynamiques, publiques et déclarées dans le type (non héritées)
pi:=tp[i].GetProperties(BindingFlags.DeclaredOnly
    or BindingFlags.Instance or BindingFlags.Public);
```

La variable `pi` est définie de la façon suivante :

```
Var pi : array of System.Reflection.PropertyInfo;
```

Chaque élément du tableau dynamique contient ainsi les informations de chaque propriété. Pour obtenir ce tableau, nous utilisons la méthode `GetProperties` de la variable indicée `tp[i]` qui contient l'objet des informations de type de la classe en cours d'inspection. Cette méthode existe en plusieurs variantes ; il existe même des variantes permettant de ne charger les informations que d'une seule propriété (par son nom, son type…). Sous la forme que nous utilisons, il faut passer à la méthode un paramètre permettant de filtrer les propriétés à charger. C'est le rôle des `BindingFlags` que vous voyez dans le code ci-dessus. Pour notre exemple, nous avons choisi de montrer les propriétés suivantes :

- `BindingFlags.Instance`, c'est-à-dire les propriétés de l'objet sans les propriétés statiques éventuelles ;
- `BindindFlags.Public`, uniquement les propriétés publiques ;
- `BindingFlags.DeclaredOnly`, uniquement les propriétés déclarées dans la classe elle-même et non celles dont elle hérite de ses parents.

Si nous obtenons une liste non vide de propriétés, nous balayons le tableau de la même façon que pour les classes et nous créons les nœuds du `treeview`.

```
if assigned(pi) and (high(pi)>low(pi)) then begin
 tt:=tvClasses.Items.AddChild(t,'Propriétés');
 for ii := low(pi) to high(pi) do begin
  if pi[ii].CanRead then s:='Read ' else s:='';
  if pi[ii].CanWrite then s:=s+' & Write';
  ttt:=tvClasses.Items.AddChild(tt,pi[ii].Name+
          ' ['+pi[ii].PropertyType.tostring+'] '+s);
  end;
```

Le code ci-dessus n'utilise qu'une partie des informations qu'on peut obtenir sur les propriétés. Il est par exemple possible de connaître le *getter* et le *setter*. Il serait trop long de détailler ici tout ce que le mécanisme de réflexion permet de connaître et nous vous

invitons à consulter la documentation du framework .NET si vous désirez en savoir plus.

Getter et Setter

On appelle ainsi les méthodes `Get` et `Set` utilisées pour lire ou modifier la valeur d'une propriété. Sous Delphi, on déclare ces méthodes spéciales au niveau de la propriété par les mots-clés `Read` et `Write`. Les méthodes sont elles-mêmes définies dans la partie `Private` de la classe le plus souvent.

D'une façon parfaitement identique à celle utilisée pour obtenir les propriétés, nous obtenons un tableau dynamique sur les méthodes de la classe en cours d'inspection. Nous le balayons de la même façon pour créer les nœuds dans le `treeview` :

```
// Méthodes publiques déclarées dans le type (non héritées)
mi:=tp[i].GetMethods(BindingFlags.DeclaredOnly
     or BindingFlags.Instance or BindingFlags.Public
     or BindingFlags.Static);
if assigned(mi) and (high(mi)>low(mi)) then begin
 tt:=tvClasses.Items.AddChild(t,'Méthodes');
 tt.ImageIndex:=-1;
 for ii := low(mi) to high(mi) do begin
  ttt:=tvClasses.Items.AddChild(tt,mi[ii].Name+
       ' ['+mi[ii].tostring+']');
  end;
 end;
```

La méthode se termine par le code suivant qui actualise l'affichage de notre fiche :

```
UpdateState; // mise à jour de l'affichage
```

Bouton Décharger

Il doit être possible de décharger le *plugin* et c'est le rôle du code derrière le bouton Décharger de s'en occuper :

```
procedure TForm1.btnUnloadPackageClick(Sender: TObject);
begin
 if assigned(fPlugin) then // si le plugin est déjà chargé
  begin
   if assigned(fObjet) then FreeAndNil(fObjet);
   FreeAndNil(fPlugin);   // déchargement
   tvClasses.Items.Clear;
  end;
 UpdateState;
end;
```

Le code ci-dessus commence par tester la variable `fPlugin` pour savoir si elle contient ou non une référence sur l'assemblage du *plugin*. Dans l'affirmative, un test de même nature est réalisé sur l'instance de l'objet qui a été créé (nous verrons cela plus bas) car il faut absolument détruire toutes les instances créées à partir des

classes contenues dans le *plugin* avant de le décharger. Enfin l'instance `fPlugin` est libérée.

Le bouton Créer Compo

Comme nous le disions dans l'introduction présentant l'application exemple, nous utilisons ici le principe le plus générique possible pour la mise en œuvre des *plugins*, c'est-à-dire que nous autorisons l'application à créer des instances de n'importe quelle classe du *plugin*. Pour ce faire, nous avons stocké dans chaque nœud de classe du `treeview` une référence vers l'objet information de type. Ainsi, il nous suffit d'inspecter le nœud sélectionné dans ce dernier pour obtenir les informations nécessaires à la création d'une instance de la classe. Comme notre *plugin* ne contient qu'un composant de test et que les autres classes contenues n'ont pas d'intérêt pour cette démonstration, nous effectuons dans le code du bouton un test sur le nom de la classe sélectionnée afin de n'autoriser que celle du composant exemple. En supprimant ce test ou en le modifiant, il est donc possible d'autoriser ou non la création de toute classe.

Le code de la création de l'instance est le suivant :

```
procedure TForm1.btnCreateComponentClick(Sender: TObject);
var tp : System.Type;
begin
 if assigned(fObjet) then exit; // si l'objet existe déjà on sort..
 if (not assigned(tvClasses.Selected)) or
    (not assigned(tvClasses.Selected.Data)) then begin
  showmessage('Sélectionnez une classe dans le treeview !');
  exit;
  end;
 tp := tvClasses.Selected.Data as System.Type;
 if CompareText(tp.Name,'TLIVREBDSPANEL')<>0 then begin
  showmessage(
   'Cette démo n''autorise la création que du Panel exemple...');
  exit;
  end;
 fObjet := Activator.CreateInstance(tp,[self]);
 TWinControl(fObjet).Parent := self; // le transtypage permet d'accéder aux
 TWinControl(fObjet).Left   := 111;  // propriétés des contrôles fenêtrés
 TWinControl(fObjet).Top    := 330;  // absente dans TComponent
 TWinControl(fObjet).Font.Size := 14;
// gestion des boutons de la fiche de l'hôte
 btnCreateComponent.Enabled := false;
 btnDestroyComponent.Enabled := true;
 btnCallAboutBox.Enabled := true;
 end;
```

Dans notre exemple, nous ne souhaitons autoriser la création que d'une seule instance du composant. C'est pour cela que la méthode commence par tester si la variable `fObjet` (de type `TObject`),

qui contient la référence de l'instance, est ou non déjà utilisée. Si l'objet est déjà créé, un message est retourné à l'utilisateur. Cette limitation est de pure circonstance et si vous supprimez ce test, il sera possible de créer autant d'instances que vous le voulez. Dans ce cas, vous utiliserez un tableau dynamique pour stocker les références de ces instances afin de les libérer toutes le moment voulu.

Un autre test permet de savoir si le nœud sélectionné dans le `treeview` est celui d'une classe ayant des informations de type ou s'il s'agit d'un autre nœud (propriétés, méthodes). Comme expliqué plus haut, ce test est complété d'une vérification du nom de la classe afin de n'autoriser que la création de notre composant exemple.

Ensuite vient le code efficace qui crée l'instance :

```
fObjet := Activator.CreateInstance(tp,[self]);
```

La création de l'instance est effectuée par l'appel de la méthode de classe `CreateInstance` de la classe `Activator` dédiée à cette tâche. Il existe plusieurs variantes de cette méthode et nous utilisons ici celle qui permet de créer une instance à partir d'une information de type en passant sous la forme d'un tableau les paramètres du constructeur de la classe. S'agissant d'un composant, le constructeur réclame une variable `Owner`, que nous passons sous la forme du paramètre `Self` de la fiche de l'application hôte.

Bouton Libérer Compo

La libération du composant est bien plus simple, comme le prouve le code ci-dessous :

```
procedure TForm1.btnDestroyComponentClick(Sender: TObject);
begin
 FreeAndNil(fObjet); // libération de l'objet et raz du pointeur
 btnCreateComponent.Enabled := true;
 btnDestroyComponent.Enabled := false;
 btnCallAboutBox.Enabled := false;
end;
```

L'instance est libérée de façon classique, le reste du code n'étant que celui de la gestion de la présentation de l'application hôte.

Le bouton À propos

Nous voulions ici vous faire voir une autre possibilité, celle d'appeler une méthode particulière d'un objet en connaissant son nom précis. Cela serait utilisé dans le cadre des *plugins* de la première catégorie, ceux qui répondent à un formalisme bien

déterminé et qui exposent des méthodes dont les noms sont connus d'avance.

C'est pour cela que nous avons ajouté une méthode `AboutBox` à notre composant. Cette méthode montre simplement un message et n'est pas d'un grand intérêt. En revanche, la façon de l'appeler est elle bien plus technique, comme le montre le code ci-dessous :

```
procedure TForm1.btnCallAboutBoxClick(Sender: TObject);
var p:System.Reflection.Methodbase;
begin
 if fObjet=nil then exit;
 p:=MethodBase(fObjet.MethodAddress('AboutBox'));
 if assigned(p) then p.Invoke(fObjet,[])
  else showmessage('Ne trouve pas "AboutBox"');
end;
```

Pour commencer, nous déclarons une variable `p` de type `System.Reflection.Methodbase`, c'est–à-dire un objet décrivant les particularités d'une méthode d'une classe.

Nous chargeons ensuite cette variable des informations de la méthode `AboutBox` par un appel à `MethodeBase`, qui prend en paramètre l'adresse de la méthode. Celle-ci est retournée en utilisant la méthode `MethodAdress` de `TObject` qui prend en paramètre le nom en clair de la méthode.

Si nous obtenons une référence valide sur les informations de la méthode, nous pouvons l'exécuter en utilisant la méthode `Invoke`. Celle-ci prend en paramètre l'instance concernée (`fObjet`) et la liste des paramètres attendus. Comme il n'y en a aucun ici, nous passons un tableau vide.

Il existe bien entendu des méthodes permettant de connaître la liste des paramètres attendus, d'analyser leur type, et de prendre connaissance de tout ce qu'on peut imaginer à propos d'une méthode… Tout cela est décrit en détail dans l'aide du framework.

Conclusion

Cet article a été l'occasion d'étudier de nombreux domaines, les packages, les `DLL`, les *plugins* et le chargement dynamique des packages, la création d'instances de classe dont on ne sait rien au départ et l'activation de méthodes dans ces instances. Les principes des mécanismes de la Réflexion du framework, équivalents des anciennes RTTI de Delphi, ont été posés et des exemples complets ont été analysés. S'il n'est certes pas dans l'ambition de l'auteur de vous faire croire que le tour de la question a été fait, il espère néanmoins qu'armé des bases ici présentées, vous serez capable

d'aborder les documentations du framework avec une plus grande sérénité et de concevoir des applications intelligentes, extensibles, ouvertes et tous ces mots qui plaisent aux commerciaux et qui procurent techniquement une grande satisfaction aux développeurs !