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

Migration de code Delphi Win32 vers .NET – Partie 2

Nous allons ici poursuivre l'étude de la migration du code Win32. Ensuite nous aborderons la migration des composants puis la façon d'importer des composants Windows Forms sous VCL.NET.

Le portage des applications Win32 – suite

Migration étape 2 : Delphi pour .NET

Arrivé ici, nous avons, d'une part, le code original dans son répertoire (code que nous n'avons pas changé) et, d'autre part, le code préparé sous Delphi 7 qui est stocké dans un autre répertoire. Une petite sauvegarde de tout cela sur un CD-Rom serait une bonne idée, qu'en pensez-vous ? …

Avant de lancer BDS, profitons de cette pause pour faire le ménage dans le répertoire de l'application en cours de migration. Supprimons allégrement tous les fichiers suivants :

- `*.dcu`, les unités compilées qui ne servent à rien pour .NET ;
- `*.~*`, les fichiers de sauvegarde du code modifié ;
- le fichier `LESDFM.TXT` dont nous n'avons plus besoin.

Conservons en revanche le fichier exécutable du projet qui est une référence bien pratique pour comparer les comportements. Comme les exécutables .NET ont aussi l'extension EXE, nous devons soit copier l'actuel exécutable soit le renommer. Nous préférons cette dernière option qui permet de laisser ce fichier dans le même répertoire. Pour faire simple, nous ajoutons le suffixe `_D7` au nom, `Demo.exe` devenant `Demo_D7.exe`.

Ouverture du projet sous Delphi .NET

Nous sommes maintenant prêts. Ouvrons Delphi .NET et chargeons le fichier projet (`DPR`). La figure 21.1 illustre le parfait déroulement des opérations. Grâce à notre préparation du code sous Delphi 7, tout se charge correctement jusque-là.

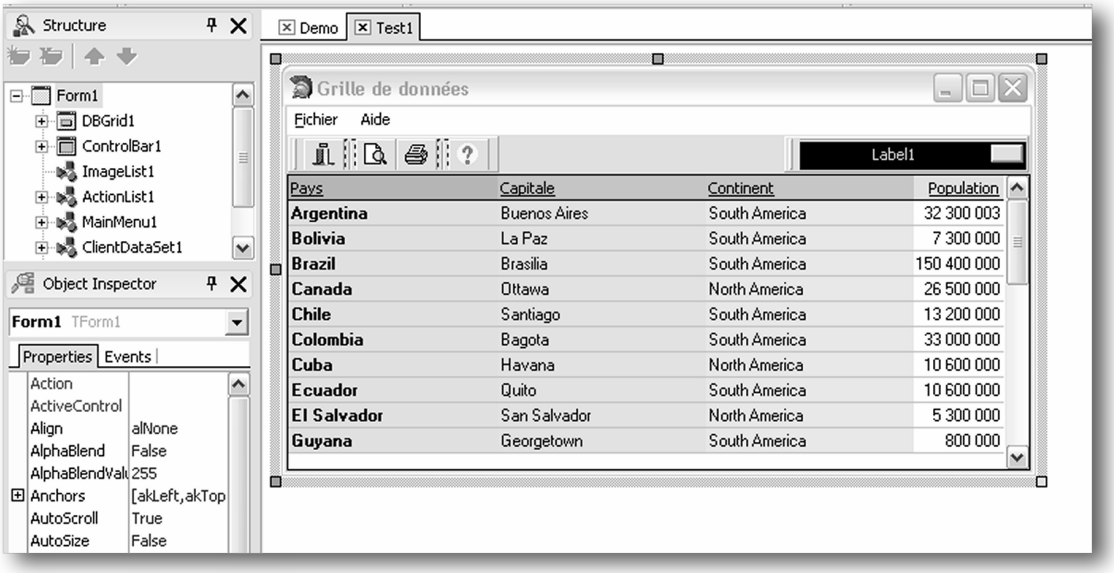

Figure 21.1

Le bon chargement de notre application Delphi Win32

Nous pouvons faire une petite pause pour admirer ce qui se cache derrière cet écran : l'énorme travail des équipes de Borland qu'il faut saluer. Oui, une application Delphi Win32 dont certains morceaux sont même hérités de version anciennes de cet EDI se charge dans un environnement natif .NET…

La première compilation

Mais trêve d'extase, il est trop tôt pour sabler le champagne. Ctrl + F9 (et oui ça marche pareil !) pour lancer une compilation et découvrir les petits soucis…

Compilation vs construction

Rappelons que dans cette phase aussi de la migration il est plus prudent de faire des constructions complètes du projet plutôt que de simples compilations. Cependant, les risques ne sont plus les mêmes que durant la phase précédente. De toute façon, si le code contient des erreurs, compilation ou construction, cela ne passera pas…

Nous découvrons toute une liste d'avertissements, mais aussi quelques erreurs fatales. Pas de panique, tout va se régler calmement. Tout d'abord, parmi les avertissements, nombreux sont ceux qui concernent les unités dépendantes de la plate-forme. Toute la VCL, même pour .NET, est dépendante de la plate-forme Windows (comme les Windows Forms d'ailleurs), nous le savons très bien. Inutile de s'encombrer de ces messages donc, sauf si nous désirons aussi rendre notre code portable sous Linux, mais c'est une complication supplémentaire que nous n'envisageons pas ici. En ouvrant les options du projet, dans les options Messages du

compilateur, nous décochons *Unité plate-forme* et relançons la compilation. On y voit déjà plus clair !

La première erreur fatale

La première erreur fatale est assez étonnante : dans l'unité qui génère le code HTML, Delphi nous indique « Opérateur non applicable à ce type d'opérande » sur la ligne suivante :

```
if DbGrid.Columns.Items[lCnt].field.Tag = 0 then
```

La propriété Tag est devenu un `Variant` et sa gestion est un peu curieuse parfois (un aveu : l'auteur déteste les variants et vous déconseille, sauf obligation expresse, de vous servir de ces fourre-tout qui peuvent contenir des choux et des navets, il vous déconseille aussi d'utiliser le `Tag` par la même occasion !).

Cette erreur est finalement la bienvenue. Elle permet de se poser la question de l'utilité de ce test. Après vérification du code, il s'avère que cela devait servir « avant » (charme du vieux code qu'on a de surcroît pas forcément écrit soi-même…) mais, en tout état de cause, à aucun endroit ce fameux Tag n'est utilisé. En toute logique, il ne peut jamais être à autre chose que zéro donc nous supprimons le test. Il ne faut pas oublier de noter cela sur un bloc-notes pour la documentation du nouveau code, et surtout pour être sûr de bien tester le fonctionnement de la partie incriminée lors de la prochaine exécution de test de notre application.

La seconde erreur fatale

Plusieurs erreurs fatales sont désormais liées à l'utilisation de la fonction `ShellExecute` dont voici le code :

```
if Preview then
   Result := ShellExecute(Handle,'Open',
             StrPCopy(St, FileName),nil,nil,SW_SHOW)
else
   Result := ShellExecute(Handle,'Print',
             StrPCopy(St, FileName),nil,nil,SW_SHOW);
```

L'une des erreurs retournées concerne la fonction `StrPCopy`, une fonction transformant une chaîne Delphi en un `PChar`. Ces derniers étant dans des pointeurs, ils n'existent pas sous .NET de cette façon-là. D'ailleurs, sous Delphi 7, nous aurions pu écrire directement `PChar(FileName)`, la conversion des chaînes dynamiques en `PChar` étant gérée automatiquement par le compilateur. Bien entendu, nous avons laissé ce vieux code Delphi dans notre exemple pour vous montrer ce que vous risquez de rencontrer. Tous les codes à migrer ne seront pas de belles applications toutes neuves écrites en pur style Delphi 7…

L'autre erreur qui est liée à `ShellExecute` est elle plus compréhensible puisque le compilateur nous indique « Aucune version surchargée de `ShellExecute` ne peut être appelée avec ces arguments ». Le simple fait qu'il y ait des « `nil` » dans notre code original (donc des pointeurs) nous indique que les paramètres ont forcément changé de type.

Tout cela se règle finalement très bien en regardant dans l'aide les fameuses différentes versions de `ShellExecute` sous VCL .NET. En fait, tout est resté identique sauf que nous avons gagné en cohérence. Le code ci-dessus devient alors :

```
if Preview then
   Result:=ShellExecute(Handle,'Open',FileName,'','',SW_SHOW)
else
   Result:=ShellExecute(Handle,'Print',FileName,'','',SW_SHOW);
```

Ce salutaire changement nous a permis aussi de supprimer la variable « `St` » qui était déclarée comme un tableau de 256 caractères (un `Pchar` statique donc). De même, nous voyons que tout est devenu plus simple : là où une chaîne est attendue, il faut mettre une chaîne et non plus s'embêter inutilement avec des conversions de type. Le framework .NET est bien plus cohérent que celui de Win32, et cette mise à égalité des langages est profitable à Delphi. Tout ce qui était lié au bricolage Delphi vers C et C vers Delphi n'existe plus. .NET est un vrai soulagement pour les développeurs Delphi.

Autre façon de faire

Dans le chapitre consacré à MyBase de l'ouvrage « Delphi 2006 et C# » chez Eyrolles nous utilisons un procédé purement .NET pour simuler le `shellexecute`. Cette façon de faire est plus propre. Consultez le code source de l'application exemple de cet article pour voir comment la même opération est réalisée en n'utilisant que les classes du framework.

Une nouvelle construction nous prouve que tout est désormais au point. En tout cas, notre application se compile. C'est peu, mais c'est énorme…

La première exécution

Cliquons sur la flèche d'exécution pour lancer l'application… Suspens ! ……Boom ! Le suspens n'aura pas été long… D'entrée de jeu nous nous prenons une exception « Modèle de thread incorrect » (`STAThreadAttribute` est obligatoire). Rien de bien méchant non plus malgré les apparences. Comme nous l'avons déjà expliqué, les applications qui utilisent les bases de données doivent comporter l'attribut personnalisé `[STAThread]` au début du premier bloc du projet. Nous allons l'ajouter juste devant le `Begin` du fichier DPR (source du projet).

Fin de la migration

Après la dernière correction, nous lançons notre application et le miracle se produit : tout fonctionne comme dans la version originale. Pourtant, il s'agit maintenant d'une application native .NET. Nous vous ferons grâce d'une nouvelle copie d'écran qui ne ferait que reproduire les figures qu'on trouve dans l'article précédent, à la différence du bouton non standard de changement de couleur qui a été remplacé.

That's all folks !

Plutôt que de passer en revue tous les composants VCL de Delphi .NET que vous connaissez déjà ou que vous pouvez découvrir aisément par vous-même, nous avons préféré réaliser « en direct » la migration d'une vieille application Delphi pour décomposer, étape après étape, ce qu'il faut faire et comment réagir aux divers petits problèmes inévitables. Cela a été réalisé « sans trucage », l'article a été écrit au fur et à mesure des manipulations. À partir des règles de migration que nous vous avons indiquées et après l'étude de ce parcours réaliste, car réel, d'une migration réussie, nous sommes certains que vous saurez aborder votre premier portage avec plus d'assurance.

Migration d'un composant

Les composants sont de magnifiques oiseaux migrateurs… Le même code écrit sous Delphi 1 (Windows 16 bits) peut aujourd'hui encore être repris pour devenir natif .NET après un voyage au-dessus du monde agité de Win32 et un détour par les plaines arides de Linux. Près de dix années de survol des plates-formes informatiques leader du marché, voyant au détour des paysages mourir bon nombre de bibliothèques concurrentes, étoiles filantes de l'univers Microsoft ou de microcosmes nombrilistes.

La VCL, applaudie par beaucoup et dénigrée par d'autres, dont les imperfections ont été souvent plus commentées que ses formidables atouts, celle que des oiseaux de mauvais augures voient mourir à chaque changement de technologie, défie à nouveau aujourd'hui les rapaces technophiles qui voudrait voir en les Windows Forms la seule façon « propre » de faire du .NET. L'oiseau migrateur s'en moque et poursuit son vol !

Grâce à la VCL, le code Delphi (toutes versions et toutes plates-formes confondues) est portable sous .NET de façon native. Cela est bien entendu vrai pour les applications mais aussi pour les

composants ! Des milliers de composants engrangeant un savoir-faire de près de dix ans de VCL sont disponibles pour répondre à des milliers de besoins, ils n'attendent qu'une chose, être migrés. Nous allons voir ici comment s'effectue une telle migration en partant d'un composant écrit il y a plusieurs années.

Le composant à migrer

Pour cette migration, nous avons choisi un composant regroupant un certain nombre de caractéristiques qui le rendent ni plus, ni moins exotique que la majorité des composants existants. Une sorte de « monsieur tout le monde » des composants qui illustrera notre propos et ressemblera un peu à tous ceux que vous aurez à migrer. Quelques caractéristiques de ce composant :

- il est visuel ;
- il hérite d'un composant de la VCL (`TShape`) ;
- il utilise des ressources internes (un `timer` ici) ;
- il expose des propriétés ;
- il est suffisamment court et simple pour que nous ne perdions pas de temps à en expliquer le fonctionnement ;
- il est un cas réel et non une création à dessein pour une belle démonstration sans problème ;
- il provient d'un vieux code (première version date du 1/6/96 !) hérité de Delphi 1 sous Win16 qui a déjà été migré sous Win32.

L'ensemble de ces points en fait un bon représentant du code typique que l'on a généralement à porter.

> **Suivez la progression avec le code source original**
> La démonstration d'une migration de composant qui est proposée ici suit une démarche pédagogique qui nous a semblée bien plus appropriée que la simple édiction d'une liste de prescriptions techniques. Pour en tirer tout le bénéfice didactique, nous vous conseillons vivement d'utiliser le code source de l'exemple sur le CD-Rom du livre et de suivre la progression pas à pas sous Delphi, le code sous les yeux.

Il s'agit d'un objet graphique dessinant une forme choisie parmi plusieurs possibilités. Pour vous faire une idée, outre le code des exemples fourni sur le CD-Rom du livre, vous pouvez simplement regarder, dans la palette d'outils Suppléments de Delphi .NET, la classe `TShape`. Notre composant reprend l'essentiel de ce dernier en lui ajoutant des comportements nouveaux que leur seule description suffit à imaginer :

- `ShapeCaption`, un texte pouvant être inclus dans la forme ;
- `ShowCaption`, une bascule activant ou désactivant l'affichage du texte ;
- `Font`, la fonte pour le texte ;
- `Alignment`, l'alignement du texte dans la forme ;
- `Blink`, une bascule activant ou désactivant le clignotement du texte ;
- `BlinkInterval`, l'intervalle de clignotement ;
- `ShadowCaption`, `ShadowColor`, `ShadowXOffset`, `ShadowYOffset`, un ensemble de propriétés gérant l'affichage, la couleur et l'offset X/Y d'une ombre portée pour le texte ;
- l'événement `OnBlink` déclenché à chaque clignotement du texte pour synchroniser le composant à d'autres affichages et ce, sans utiliser un autre timer.

Ce composant a été intégré dans une application de test sous Delphi 7 afin d'en vérifier le fonctionnement. Cette application sera elle aussi portée sous Delphi .NET et servira le même but. Nous ne montrerons pas sa migration, le portage d'une application ayant été exposé dans l'article précédent.

Étape 1 – Préparation sous Delphi 7

Nous ne le répéterons certainement jamais assez, ne travaillez jamais sur le code original pour une migration, faites-en une copie complète dans un répertoire séparé et travaillez sur cette copie.

Appliquant nos propres conseils, nous avons copié le code dans un répertoire de migration avec le suffixe « D7 » puisque dans un premier temps, nous effectuerons les mêmes vérifications et modifications sur le code de notre composant que sur une application standard. Nous ne détaillerons pas plus cette phase largement commentée dans l'article précédent (notons qu'elle peut être réalisée sous Delphi 7 ou sous la personnalité Delphi Win32 de BDS).

Notre composant a ainsi passé sa première étape de migration avec succès, il ne contenait rien qui puisse nous obliger à en modifier le code. La compatibilité est donc maximale ici.

Étape 2 – premier chargement sous Delphi .NET

Notre composant original n'a pas été modifié, nous l'avons copié et contrôlé sous Delphi 7 à partir d'une copie. Nous allons conserver ces deux versions fonctionnelles et créer un nouveau répertoire définitif.

Nous ouvrons le fichier package de notre composant (`.dpk`), les éléments de notre composant sont visibles dans le gestionnaire de projet.

Étape 3 – Modification de l'icône

Dans le gestionnaire de projet, nous supprimons le fichier icône (`.dcr`) du projet. En effet, les composants Delphi .NET n'utilisent plus ce fichier (qui n'était qu'un fichier ressource Windows renommé de `.res` à `.dcr`) mais un fichier bitmap séparé pour chaque composant.

La première vraie modification de notre code consistera donc à modifier… son icône !

Les nouvelles règles pour l'icône d'un composant Delphi pour .NET sont les suivantes :

- L'image doit être en 16 x 16 pixels et non plus en 24 x 24.
- Elle doit être au format bitmap (`.bmp`) en 16 ou 256 couleurs.
- Le nom du bitmap doit être : `<nomunité>.<nomclasse>.bmp`.
- Les noms de l'unité et de la classe sont sensibles à la casse et doivent être écrits comme dans les déclarations de l'unité du composant.
- La ressource bitmap doit être liée au composant en plaçant dans l'unité de celui-ci une commande `{$R '<nomdubitmap>'}`.

En règle générale, le passage de 24 x 24 en 16 x 16 pixels interdit de simplement retailler l'icône avec un logiciel de dessin car cette diminution rend l'image illisible. Il faut donc refaire l'icône proprement. Toutefois, si vous désirez reprendre l'ancienne icône avant de supprimer le fichier `.dcr`, voici ce qu'il faut faire :

1. Charger le programme imagedit.exe qui se trouve dans le répertoire `/bin` de Delphi.

2. Ouvrir le fichier `.dcr`.

3. Extraire le bitmap (ou les bitmaps s'il s'agit d'une bibliothèque de composants) et le sauvegarder sur disque dans le répertoire du composant.

4. Modifier son nom en appliquant les règles de nom indiquées plus haut.

5. Supprimer le `.dcr`.

6. Ajouter dans le code du composant : `{$R <nombitmap.bmp>}`.

Étape 4 - Modification du package

Les seules modifications essentielles du package concernent la liste des `requires` (`Requiert` dans le gestionnaire de projet, `requires` dans le code source du package).

Dans notre cas, nous allons supprimer les références aux unités Win32 de la VCL telles : `rtl.dll` et `vcl.dll`. Nous allons les remplacer par les références au framework VCL .NET : `Borland.Delphi`, `Borland.Delphi.VclRtl` et `Borland.vcl`. Notre package va donc devenir dépendant de ces derniers.

Nous lançons la compilation du package. Si aucun avertissement ni aucune erreur n'est signalé, le composant est donc prêt à être installé.

Il est possible de pratiquer un certain nombre d'autres aménagements dans le code source du projet. Nous ne vous conseillons pas de supprimer ou de modifier les options de compilations en début du fichier `.dpk`, le code étant totalement portable avec Delphi Win32 de la façon dont nous avons procédé il serait dommage de se priver de cette versatilité en supprimant ce qui est utile à Delphi 7 et qui ne gêne pas Delphi .NET.

Nonobstant, vous pouvez au moins ajouter une description plus complète, comme la version de votre assemblage. Vous l'avez certainement remarqué, le package compilé de Delphi .NET est une DLL… donc tout ce qui s'applique à ce format est valable pour un package de composants. Ainsi, vous pouvez placer derrière la clause `contains` du projet package les lignes suivantes :

```
[assembly: AssemblyTitle('Titre de l''assemblage')]
```

```
[assembly: AssemblyDescription('Description de l'assemblage')]
[assembly: AssemblyConfiguration('Configuration de l'assemblage')]
[assembly: AssemblyCompany('Votre société')]
[assembly: AssemblyProduct('Nom du produit')]
[assembly: AssemblyCopyright('Vos copyrights')]
[assembly: AssemblyTrademark('Votre marque déposée')]
[assembly: AssemblyCulture('')]        // gestion du langage de .NET
[assembly: AssemblyVersion('7.1.*')]   // gestion de version
[assembly: AssemblyDelaySign(false)]   // signature non différée
[assembly: AssemblyKeyFile('')]        // fichier clé de signature
[assembly: AssemblyKeyName('')]        // nom de la clé de signature

// la description ancienne version qui doit déjà se
// trouver dans votre package
{$DESCRIPTION 'Mon package à moi'}
```

On peut voir dans le code ci-dessus quelques entrées propres à la signature électronique de l'assemblage, nous aborderons ce sujet dans un chapitre de l'ouvrage « Delphi 2006 et C# » paru en mars 2006 chez Eyrolles.

Étape 5 – Installation du composant

Pour installer un nouveau composant, vous devez ouvrir le menu Composant puis sélectionner « Composants .NET installés ». Un dialogue à plusieurs onglets s'ouvre alors et propose en première page les composants .NET du framework (onglet Composants .NET). Il faut cliquer sur l'onglet « Composant VCL .NET » pour installer notre composant. Après avoir cliqué sur le bouton Ajouter, nous parcourons les répertoires pour sélectionner le fichier `.dll` de notre package (qui est dans le répertoire de notre composant). L'affichage ressemble alors ce que vous pouvez voir figure 21.2.

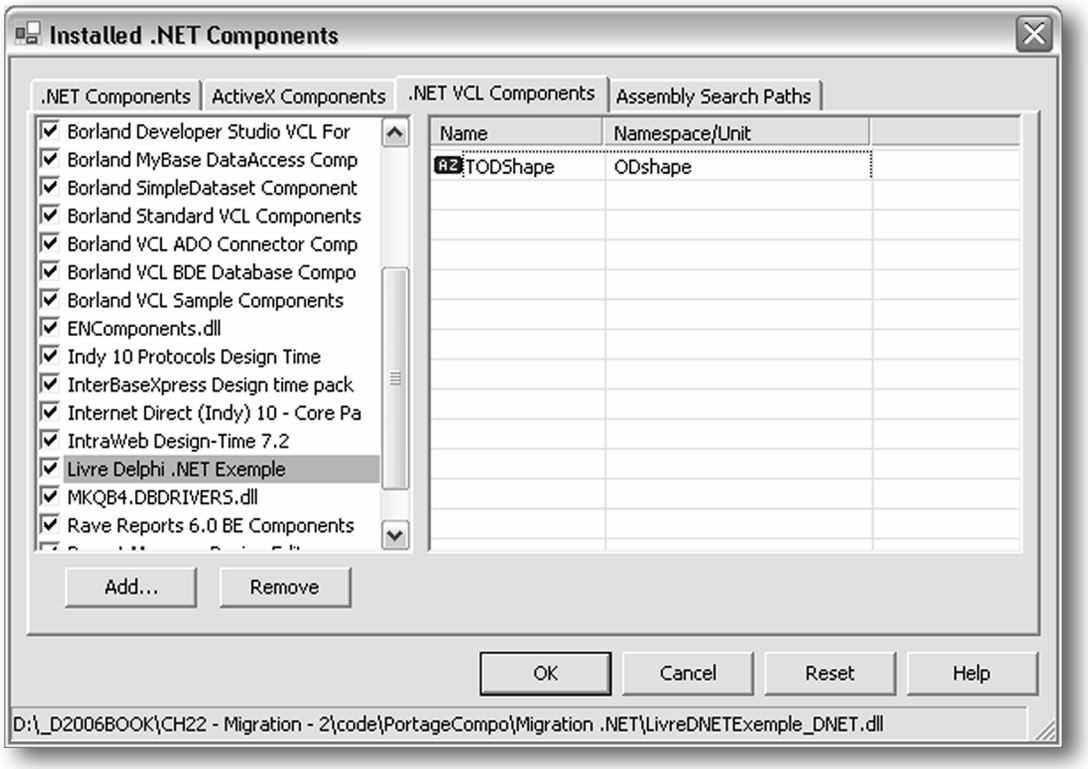

Figure 21.2

L'ajout d'un nouveau package de composant VCL .NET

Étape 6 – Vérification de l'installation

Arrivé à la fin de l'étape 5, notre nouveau composant est installé dans l'EDI de Delphi .NET. Pour nous prouver que tel est bien le cas, nous allons fermer tous les fichiers ouverts et démarrer un nouveau projet VCL vierge.

En balayant la palette d'outils (en bas à droite de l'EDI par défaut), nous constatons que notre palette a bien été installée (figure 21.3).

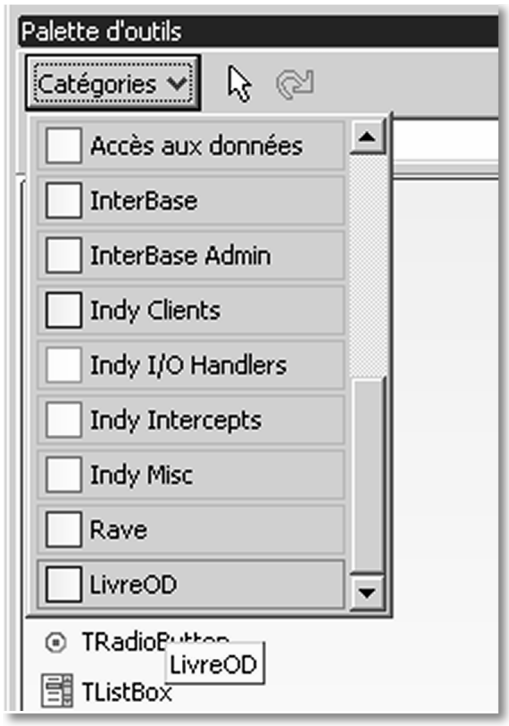

Figure 21.3

La palette « LivreOD » est bien présente dans la palette d'outils

Si nous sélectionnons la palette « LivreOD » et obtenons la confirmation qu'elle contient effectivement notre composant `TODShape` (figure 21.4).

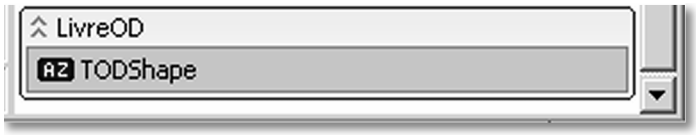

Figure 21.4

Le nouveau composant TODShape est bien recensé

Nous remarquons par la même occasion que la nouvelle icône du composant a elle aussi été bien intégrée et qu'elle est reconnue par l'EDI.

Étape 7 – Chargement de l'application de test

Il serait en effet bien naïf de croire que la seule présence de notre composant dans les palettes d'outils de Delphi est une preuve définitive de son bon fonctionnement… Notre code a passé l'étape de la compilation, il a pu être reconnu par l'EDI, son icône s'affiche et son nom est présent dans les palettes. Tout cela est

positif mais ne doit pas nous faire oublier qu'un code n'est validé que s'il a passé tous les tests fonctionnels conformes au plan de test établi lors de sa conception. S'agissant ici d'un modeste composant, notre plan de test consiste au moins à vérifier que tout fonctionne comme prévu en utilisant l'application de test conçue sous Delphi Win32 et qui permet de manipuler les principales propriétés du composant.

La migration de cette application repose sur les mêmes mécanismes que ceux présentés dans l'article précédent, nous n'en dirons pas plus ici. Dans notre cas, le portage a été direct et nous n'avons pas eu à modifier le code de l'application. La figure 21.5 montre l'application de test chargée dans l'EDI de Delphi .NET. Notre composant, posé sur la partie droite de la fenêtre, s'affiche correctement, nous sommes sur la bonne voie…

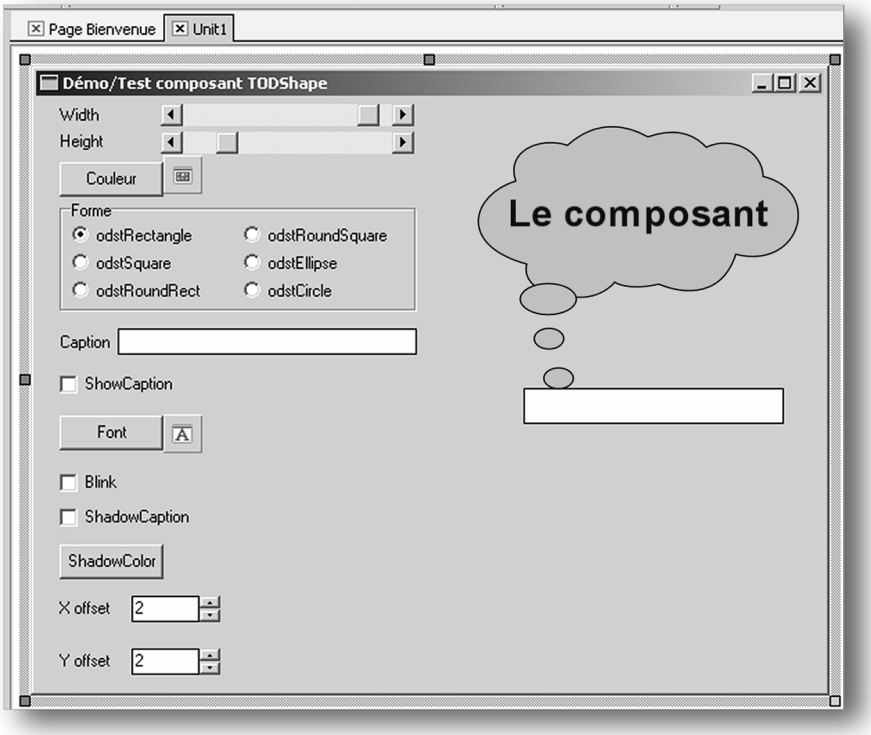

Figure 21.5

La fenêtre de l'application de test sous EDI Delphi .NET

Il ne reste plus qu'à exécuter l'application et jouer avec les réglages qu'elle propose pour voir notre composant s'animer. Les livres n'étant pas encore publiés sur du *e-paper*, vous ne pourrez pas voir ici le texte clignoter. Cela étant, vous pouvez vous en assurer en utilisant le code complet de cet exemple qui se trouve sur le CD-Rom du livre.

Les packages Delphi pour .NET

Comme l'exemple de migration de composants présentée plus haut l'a montré, les packages Delphi ont subit une légère modification en passant sous .NET puisqu'ils sont devenus des DLL.

En fait, rien de nouveau puisque les packages des versions précédentes étaient aussi des DLL mais avec une extension `.bpl`. Ce changement se justifiait pour une raison fort simple : sous Win32, ni les API, ni le fonctionnement global du système d'exploitation n'étaient objet. Ainsi, les DLL ne pouvaient qu'exporter des ressources et des fonctions. Delphi, pour tirer avantage de sa structure objet, avait modifié le format DLL standard pour permettre d'y stocker des classes (généralement des composants VCL) ce qui justifiait le changement d'extension de fichier. Les packages Delphi Win32, pour être plus puissants que les DLL standards, n'en perdaient pas moins toute espérance de compatibilité avec les autres langages, avantage majeur des DLL.

Sous .NET, où, à la principale différence de Win32, tout est objet, les DLL sont des assemblages similaires aux exécutables `.exe`. Elles contiennent en effet tout ce qu'il faut pour exporter des classes de manière universelle (puisque tous les langages .NET sont compatibles entre eux). De fait, Delphi n'a plus besoin de se créer son propre format de package pour y stocker des classes, il lui suffit d'utiliser le format standard .NET des DLL. Au passage, les packages de Delphi retrouvent l'universalité perdue avec le format `.bpl` et ses classes peuvent aujourd'hui être utilisées par tous les langages .NET. Il faut moduler cette assertion en rappelant que la VCL est propre à Borland et que tout package de composants au format VCL ne pourra vraisemblablement pas être utilisé par d'autres langages. Cependant, le format des packages n'est pas réservé à la publication de composants VCL, il en est même totalement indépendant et peut donc être utilisé sous Delphi pour partager des classes avec tous les autres langages de la plate-forme.

Les packages peuvent être scindés en deux grandes familles : les packages d'exécution (*runtime packages*) et les packages de conception (*design time packages*). Les premiers sont utilisés pour stocker du code exécutable, les seconds sont à destination de l'EDI de Delphi et contiennent le code qui lui est spécifique comme les éditeurs de propriétés, les éditeurs de composants, les experts…

La motivation

Il existe plusieurs raisons objectives d'utiliser les packages.

Réduire la taille du code

Lorsque l'on utilise des packages runtime, Delphi n'associe plus le code des composants à celui de votre EXE. Ce dernier devient donc beaucoup plus petit. Par contre, vous devez fournir les packages compilés en accompagnement de votre logiciel. Comme le *linker* de Delphi est intelligent (c'est un *smart linker*), il supprime le code non appelé : un EXE avec les packages de runtime semblera plus petit mais l'ensemble des fichiers à fournir sera bien plus gros que l'EXE 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 EXE sur un même site client. À partir d'un moment, plein de petits EXE partageant les mêmes packages finissent par être bien moins gros que plusieurs EXE intégrant à chaque fois tout le code nécessaire.

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

La séparation du code en unités indépendantes (les packages) peut toutefois être avantageuse, 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 EXE 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 le framework .NET propose d'autres mécanismes permettant le téléchargement partiel d'une application via l'Internet et il peut être intéressant dans certains cas de suivre ce nouveau modèle.

Englober un ensemble de composants

Si vous désirez distribuer un ensemble de composants (même un seul d'ailleurs) 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 systématiquement par celle de packages ! Les composants autonomes « à la Delphi 1 et 2 » n'existent plus, leur installation depuis Delphi 3 passe par leur intégration dans un package utilisateur, ce que propose l'expert dédié à cet effet dans l'IDE. Mais nous vous conseillons de bien séparer vos composants en packages différents que vous créerez vous-même plutôt que de tout ajouter dans le fourre-tout du package utilisateur proposé par l'EDI.

Le partage de classes avec les autres langages .NET

Sous .NET, Delphi est plus universel qu'il ne l'a jamais été, sachez en tirer profit ! Les packages Delphi pour .NET sont des assemblages tout à fait standard et les classes qu'ils exportent peuvent être utilisées par les autres langages .NET.

La création des plug-ins

Les plug-ins sont aux logiciels ce que le bus d'extension est aux PC : un moyen extraordinaire de laisser à d'autres le soin d'étendre les possibilités de votre création et de se faire, pourquoi pas, de l'argent tout en faisant la promotion de votre propre travail… Le PC a conquis le monde en partie grâce à sa possibilité d'y ajouter des cartes. Si vous regardez aujourd'hui les logiciels à succès, vous verrez qu'ils sont souvent dotés d'une telle possibilité d'extension via du code externe. Photoshop d'Adobe ne serait certainement pas ce qu'il est sans tous les éditeurs qui lui ont écrit des filtres, Cubase de Steinberg ne serait certainement pas le standard qu'il est chez les musiciens s'il n'était pas doté des extensions VST permettant d'ajouter des effets spéciaux et des simulations de synthétiseurs… Les exemples ne manquent pas et grâce aux packages, vous pouvez faire de même pour vos logiciels.

Les différents types de packages

Il existe quatre types de packages qui peuvent être créés en combinant les options de compilation ad hoc. Comme nous le verrons, certains types sont plus utilisés que d'autres (voir tableau 21.1).

Tableau 21.1

Les différents types de packages

TYPE DE PACKAGE	UTILITÉ
Runtime	Ces packages contiennent des classes et des composants. Ils sont nécessaires à l'exécution des applications qui s'en servent et ils doivent être déployés.
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 de vos applications et n'ont pas à être déployés avec ces dernières.

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…). Un tel package sert aussi bien à compiler des applications autonomes que des applications l'utilisant dynamiquement.
Non Designtime ni Runtime	Voilà des bêtes bien curieuses ! C'est une espèce excessivement rare de package. Leur unique vocation est d'être exploitée par des références directes depuis une application ou depuis l'environnement de conception. De tels packages peuvent être intégrés dans d'autres packages (bibliothèques de fonctions par exemple).

Les fichiers des packages

Les packages sont le fruit de la fusion de différents fichiers, leur compilation génère d'autres fichiers et tout cela ne semble pas toujours très clair. Fixons alors les choses avec le tableau 21.2 qui liste l'ensemble des fichiers liés directement à la notion de package.

Tableau 21.2

Les fichiers impliqués dans un package

EXTENSION	TYPE	DESCRIPTION
`.dpk`	Source	**D**elphi **P**ac**K**age Ce fichier est créé par Delphi lorsque vous appelez l'éditeur de packages. Le `.dpk` est l'équivalent pour les packages du `.dpr` des projets standard. Il contient une description des unités plutôt que le code utile qui est stocké dans des unités liées.
`.dcpil`	Code compilé	**D**elphi **C**ompiled **P**ackage for **I**ntermediate **L**anguage Ce fichier contient la version compilée en MSIL 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	DLL au format .NET C'est le fichier runtime ou design time du package équivalent à une DLL Win32. Si le package est de type runtime, c'est ce fichier que vous distribuerez avec votre 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 distribuer le fichier `.dcpil` avec le fichier DLL.

Directives de compilation des packages

Les packages peuvent être de différents types comme nous l'expliquions plus haut. Ces types sont matérialisés dans le code source du projet package par différentes directives de compilation. Le tableau 21.3 donne la liste des directives propres aux packages.

Tableau 21.3

Les directives de compilations propres aux packages

DIRECTIVE	LOCALISATION	DESCRIPTION
{$G-} ou {$IMPORTEDDATA OFF}	Unité	La directive {$G-} désactive la création de références aux données importées. {$G-} et empêche l'unité paquet dans laquelle elle apparaît de faire référence à des variables se trouvant dans d'autres paquets.
{$DENYPACKAGEUNIT ON}	Unité	La directive {$DENYPACKAGEUNIT ON} empêche l'unité dans laquelle elle apparaît d'être placée dans un package.
{$DESIGNONLY}	Projet (dpk)	Directive de package. Compile en mode Design seulement.
{$RUNONLY}	Projet (dpk)	Directive de package. Elle compile en mode runtime seulement.
{$IMPLICITBUILD OFF}	Projet (dpk)	Elle empêche le package d'être reconstruit implicitement par Delphi lors d'une reconstruction. Utilisez cette directive dans les fichiers DPK lors de la compilation de paquets qui fournissent des fonctionnalités de bas niveau et qui ne sont pas modifiés fréquemment ou dont le source n'est pas distribué.
{$DESCRIPTION 'texte'}	Projet (dpk)	Elle permet de donner une description textuelle à un paquet. Cette ligne sera affichée dans la fenêtre de gestion des packages de l'EDI. N.B. : la chaîne doit faire moins de 256 caractères. Sous .NET, on peut utiliser aussi les directives de la migration d'un composant montrées à l'étape 4 précédemment.

Installer un package

Pour avoir une description pas à pas de l'installation d'un package dans l'IDE de Delphi, reportez-vous à l'étape 5 de notre exemple de migration de composant en début de cet article.

Vous noterez qu'en dehors de la méthode qui est exposée, un package peut aussi être installé en utilisant l'option Installer du menu local dans le gestionnaire de projet lorsque le projet package y est chargé.

Équivalences des packages Win32

Si l'exemple de migration de composant en début de cet article donne déjà une vision assez claire du portage d'un package, il convient de compléter les informations que nous avons fournies, notamment à propos des changements à effectuer dans la partie `requires` des sources du package (`.dpk`).

En effet, les anciennes références aux unités de la VCL doivent être supprimées et remplacées selon les équivalences données par le tableau 21.4.

Tableau 21.4

Équivalences des noms de packages Borland

ANCIENNE REFERENCE	NOUVELLE REFERENCE .NET
`Rtl`	`Borland.Delphi` et `Borland.VclRtl`
`Vcl`	`Borland.Vcl`
`Vclx`	`Borland.VclX`
`Dbrtl`	`Borland.VclDbRtl`
`Bdertl`	`Borland.VclBdeRtl`
`Vcldb`	`Borland.VclDbCtrls`
`dbexpress`	`Borland.VclDbExpress`
`dbxcds`	`Borland.VclDbxCds`
`Dsnap`	`Borland.VclDSnap`
`dsnapcon`	`Borland.VclDSnapCon`
`vclactnband`	`Borland.VclActnBand`
`IBXpress`	`Borland.VclIBXpress`

Unités implicitement liées

Lorsqu'une unité de code est utilisée par un package sans être listée dans sa clause `contains`, le compilateur affiche l'erreur « unité liée implicitement » au moment de la compilation.

La façon de remédier à la situation est de veiller à ce que toutes les unités utilisées par le package apparaissent bien dans la liste `contains` du projet (`.dpk`).

Dans certaines circonstances, l'avertissement peut être considéré comme justifié sans pour autant constituer une erreur. Plutôt que de paraphraser Borland, citons directement la note suivante qu'ils ont émise :

Remarque

Sachez qu'il y a des situations où l'avertissement « Unité liée implicitement » est acceptable, par exemple quand vous construisez votre propre assemblage d'exécution avec un mélange d'unités VCL et de vos propres unités personnalisées. Cependant, les restrictions concernant l'utilisation d'un tel package/assemblage devant être installé dans l'EDI n'ont pas changé : vous ne pouvez pas construire un package/assemblage qui lie à lui-même tout .dcu/.dcuils fourni par Borland ou un package/assemblage on fourni par Borland qui le fait, et le faire fonctionner correctement dans l'EDI.

Conclusion

Ici se terminent les deux articles consacrés à la migration du code, des applications et des composants VCL Win32 vers Delphi.NET. Au-delà de l'aspect récupération de code, qui est malgré tout essentiel quand on a choisi un langage offrant une pérennité à son travail, cette étude peut aussi être vue comme un prolongement des chapitres consacrés au langage. En effet, écrire une nouvelle application sous Delphi.NET c'est quelque part pour le développeur faire, dans sa tête, la migration de ses techniques de développement…